UNIVERSITÀ DEGLI STUDI DI GENOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA
IN
INGEGNERIA INFORMATICA

TESI DI LAUREA

STRUMENTI PER LA GENERAZIONE AUTOMATICA DI TEST STRUTTURALI E FUNZIONALI

Relatore accademico
Prof. Enrico Giunchiglia

Correlatori
Dott. Emanuele Di Rosa
Prof. Massimo Narizzano

Candidato
Maurizio Pianfetti

22 Marzo 2013

Anno Accademico 2011 – 2012

TOOLS FOR TEST GENERATION IN BLACK-BOX AND WHITE-BOX TESTING

Abstract

In this thesis, applying the theory of software engineering, we analyzed and experimentally evaluated tools that allow to increase the automation of the software testing process.

This thesis is a joint work with ETT s.r.l., software house in Genoa, where we have analyzed and tested a number of projects, web applications designed and maintained by the company. The main goal of this thesis is to improve and speed-up the internal tasks related to the software testing process, thus saving time and the associated costs.

We faced both white-box and black-box testing. Concerning white-box testing, we applied Microsoft Pex, a tool for the automatic generation of unit tests, on the source code of different projects. We applied this tool on different benchmarks and compared it with some research tools developed by the research group at STAR-LAB, a laboratory of the University of Genoa (Prof. Enrico Giunchiglia, Prof. Massimo Narizzano, Dr. Emanuele Di Rosa and Dr. Gabriele Palma) and other available research tools.

Concerning black-box testing, we designed and developed a tool that automatically generate and execute tests for web applications. We considered real software applications developed at ETT s.r.l. that were already tested and delivered; thanks to the automatic tools we used, we managed anyway to find some small bugs and still improve the quality of the final product.

Dr. Emanuele Di Rosa works at ETT s.r.l. and was the company tutor of this thesis; he helped to improve the work and led the analysis and the comparison of the experimental tools. He supported also the design of the tool we developed to automatically generate tests for web applications.

Alla Commissione di Laurea e di Diploma
Alla Commissione Tirocini e Tesi

Sottopongo la tesi redatta dallo studente Maurizio Pianfetti dal titolo “Strumenti per la generazione automatica di test strutturali e funzionali”.

Ho esaminato, nella forma e nel contenuto, la versione finale di questo elaborato scritto, e propongo che la tesi sia valutata positivamente assegnando i corrispondenti crediti formativi.

Il Relatore Accademico

Prof. Enrico Giunchiglia

Ringraziamenti

Un ringraziamento penso che sia dovuto a tutte le persone che mi hanno sostenuto moralmente, a partire dagli attuali colleghi ed ex colleghi di lavoro che ad ogni esame superato pretendevano un pranzo offerto. Un ringraziamento alla mia compagna Mariangela che ha dovuto subire l'impegno da me preso trascurandola. Un pensiero al mio caro nonno Carlo che non ha avuto il tempo di sapere che finalmente ho finito gli studi, alla mia cara nonna Lorenzina ed a tutta la mia famiglia. Infine alla mia cara mamma che ha sempre creduto in me a cui dedico questa Laurea Specialistica.

Mi sono iscritto a questo corso di Laurea Specialistica pensando che potevo conciliare lavoro e studio. È stata molto dura ma penso di avercela fatta con grande soddisfazione personale, quindi un grazie anche a me stesso.

Glossario, Acronimi e Simboli

Glossario, Acronimo, Sigla	**Significato**
ETT s.r.l.	“Electronic Technology Team”, letteralmente “elettronica, tecnologia e squadra”, società genovese dove si è svolto il tirocinio
STAR-LAB	“Systems and Technologies for Automated Reasoning Laboratory”, letteralmente “laboratorio di sistemi e tecnologie per il ragionamento automatico”
Pex	“Program Exploration”, letteralmente “esploratore di programma”, strumento per l’esecuzione di test strutturali
Web	Letteralmente “ragnatela” è diventata una parola comune nell’indicare i collegamenti tra diverse applicazioni funzionanti da remoto e su Internet
WatiN	“Web Application Testing In .Net”, letteralmente “testing di applicazioni web con .NET”, libreria per l’esecuzione di test funzionali
IEEE	“Institute of Electrical and Electronics Engineers”, letteralmente “istituto degli ingegneri elettrici ed elettronici”
input	Letteralmente immettere, normalmente si indicano i dati in ingresso di qualcosa, un test o un programma
output	Letteralmente emettere, normalmente si intende un risultato in uscita di una elaborazione
XP	“eXtreme Programming”, letteralmente “programmazione estrema”
TDD	“Test Driven Development”, letteralmente “sviluppo guidato dalle verifiche”
Test	Letteralmente prova, indica una prova di utilizzo di ciò che si vuole testare
Unit Test	Letteralmente “Unità di prova”, indica il test di una specifica porzione di codice dell’applicazione testata
Test Case	Caso di studio di test
Test Suite	Insieme di casi di studio di test
$\subset$	Sottoinsieme
$\in$	Appartiene
$\notin$	Non Appartiene
$\Leftrightarrow$	Equivalenza matematica, condizione necessaria e sufficiente, corrispondenza biunivoca, se e solo se
$\exists$	Esiste
$\wedge$	Operatore logico AND, letteralmente e
$\vee$	Operatore logico OR, letteralmente oppure
$\neg$	Operatore logico NOT, letteralmente non
$\Rightarrow$	Se … Allora, quindi

IDM	“Input Domain Model”, letteralmente “modello del dominio degli input”
ISP	“Input Space Partitioning”, letteralmente “spazio dei dati di input Partizionati”
framework	Letteralmente struttura, normalmente si indica un insieme di libreria da utilizzare in un linguaggio di programmazione
assert o assertion	Letteralmente affermazione, nei test viene usata per definire se l’esecuzione ha portato ad un risultato positivo o negativo
XML	“eXtensible Markup Language”, letteralmente “linguaggio marcatore estensibile”, consente di definire e controllare gli elementi contenuti in un documento di testo.
DLL	“Dynamic Link Library”, letteralmente “libreria a collegamento dinamico”
JavaScript	Linguaggio di programmazione utilizzato nelle pagine web come client side, dove le operazioni di calcolo vengono fatte sulla macchina del visitatore e non sul server dove risiede l’applicazione web
HTML	“HyperText Markup Language”, letteralmente “linguaggio a marcatori per ipertesti”, usato per i documenti ipertestuali è un linguaggio per la creazione di pagine web
CBMC	“Bounded Model Checker for ANSI-C and C++”, modello formale per l’analisi di grafi di controllo per l’analisi statica del codice sorgente
solver	Letteralmente risolutore, colui che risolve un sistema di vincoli
SAT	“Satisfiability Arithmetic Theories solver”, letteralmente “risolutore di soddisfacibilità teorica aritmetica”
namespace	Letteralmente “spazio di nomi”, indica il nome della libreria in cui una classe C# è stata dichiarata
PUT	“Parameterized Unit Test”, letteralmente “unità di test parametrizzata”
database	Letteralmente “base di dati” si indica l’applicazione che gestisce i dati di altre applicazioni
Moles	Microsoft Moles, framework utilizzato da Pex per poter isolare il le unit test create
MSDN	“MicroSoft Developer Network”, letteralmente “rete di sviluppo MicroSoft”
DSE	“Dynamic Simbolic Execution”, letteralmente “esecuzione simbolica dinamica”
default	Si intende una impostazione o valore di base utilizzata se non ne viene impostata una
API	“Application Programming Interface”, letteralmente “interfaccia di programmazione di un'applicazione”
MSIL	“MicroSoft Intermediate Language”, letteralmente “linguaggio MicroSoft intermedio”
SMT	“Satisfiability Modulo Theories”, letteralmente “soddisfacibilità teorica di un modulo”

warning	Messaggio di avviso, potrebbe essere considerato in certi casi un errore ed in altri una informazione
log	Log, letteralmente “tronco di legno”, nell’informatica i file di log sono file di testo dove vengono salvati messaggi descrittivi sulla cronologia degli eventi dell’applicazione
browser	Applicazione per poter visualizzare pagine web
PDF	“Portable Document Format”, letteralmente “formato di documento portabile”
SOAP	“Simple Object Access Protocol”, letteralmente “protocollo per l’accesso a semplici oggetti”
REST	“REpresentational State Transfer”, letteralmente “rappresentazione dello stato traferito”
HTTP	“HyperText Transfer Protocol”, letteralmente “protocollo di trasferimento di testo dinamico”
CSS	“Cascading Style Sheets”, letteralmente “fogli di stile a cascata”
STA	“Single Thread Appartment”, letteralmente “modalità singolo thread”
MTA	“MultitTreaded Apartments”, letteralmente “modalità con più di un thread”
form	Modulo di compilazione in una pagina web
COL	Comunicazioni obbligatorie On Line
CAPTCHA	“Completely Automated Public Turing test to tell Computers and Humans Apart”, letteralmente “test di Turing pubblico e completamente automatico per distinguere computer e umani”
MVC	“Model View Controller”, letteralmente “modello, interfaccia e controllore”

Prefazione

In questa tesi sperimentale, applicando la teoria dell'ingegneria del software, sono stati analizzati e valutati sperimentalmente strumenti che permettono di aumentare l'automazione del processo di testing delle applicazioni.

Questa tesi è il lavoro congiunto con ETT s.r.l., società di informatica di Genova, dove si sono analizzati alcuni progetti e testate diverse applicazioni web progettate e mantenute dall'azienda. L'obiettivo è stato quello di voler migliorare e velocizzare le attività interne relative al processo di testing del software, risparmiando così tempo e costi connessi.

Si sono affrontate due tipologie di testing: funzionale e strutturale.

Per quanto riguarda i test strutturali, è stato utilizzato Microsoft Pex, uno strumento per la generazione automatica di unit test. Si è analizzato il codice sorgente di diversi progetti. È poi stato utilizzato lo questo strumento per l'analisi di sorgenti banchmark, confrontando i risultati con quelli di alcuni strumenti sperimentali sviluppati dal gruppo di ricerca presso STAR-LAB, laboratorio dell'Università degli Studi di Genova (Prof. Enrico Giunchiglia, Prof. Massimo Narizzano, Dott. Emanuele Di Rosa e Dott. Gabriele Palma).

Per quanto riguarda i test funzionali, è stato progettato e sviluppato uno strumento che genera ed eseguire automaticamente test per le applicazioni web. Sono state prese in considerazione le applicazioni software sviluppate da ETT s.r.l. che erano già in produzione; grazie agli strumenti automatici, si è riusciti a trovare comunque alcuni piccoli difetti che una volta corretti potranno migliorare la qualità del prodotto finale.

Il Dott. Emanuele Di Rosa dipendente presso ETT s.r.l. è stato il referente aziendale di questa tesi. Ha contribuito a migliorare il lavoro come guida per l'analisi e il confronto degli strumenti sperimentali ed ha inoltre supportato nella progettazione e sviluppo dello strumento per generare automaticamente test per le applicazioni web.

Sommario

Glossario, Acronimi e Simboli I

Prefazione IV

CAPITOLO 1 Introduzione 1

1.1. Motivazioni 1

1.2. Obiettivi 3

1.3. Organizzazione della Tesi 3

CAPITOLO 2 Il Testing del software 5

2.1. Accenni ai modelli di sviluppo 5

2.1.1. Descrizione dei modelli di sviluppo 6

Modello a cascata 6

Modello evolutivo 7

Modello a spirale 7

Metodologie agili 8

2.2. Verifica e convalida 9

2.2.1. Verifica statica 10

2.2.2. Verifica dinamica 11

2.3. Tipologie di Testing 11

2.3.1. Unit Testing 12

2.3.2. Test di Regressione 13

2.3.3. Test di Integrazione 13

2.3.4. Test di Sistema 13

2.3.5. Test di Accettazione 14

2.4. Criteri e determinazione dell'insieme di Test 14

2.4.1. Criteri di selezione 14

2.4.2. Criteri fondamentali 16
2.4.2.1. Black-Box Testing 17
Partition Testing 17
Error based Testing 21
2.4.2.2. White-Box Testing 22
CAPITOLO 3 Strumenti per la generazione automatica di Test strutturali 25
3.1. Strumenti sperimentali e commerciali 25
3.1.1. Prodotti commerciali e open source 25
NConer Desktop 25
JetBrains dotCover 27
Microsoft Pex 27
3.1.2. Strumenti sperimentali 27
CHESS 28
3.1.2.1. Software sperimentali sviluppati in ambito di ricerca 28
TeGeVe 29
noPref 29
SAT&PREF 30
SAGE 30
3.2. Microsoft Pex, strumento di analisi per la creazione di Test automatici 31
3.2.1. Struttura applicativa 32
3.2.2. Funzionamento dell'analisi effettuata sul codice da Pex 33
3.2.2.1. Esempio di esplorazione 33
3.2.2.2. Esecuzione simbolica dinamica 36
3.2.2.2.1. Instrumentation 37
3.2.2.3. Strategie 39
3.2.2.3.1. Strategia Fitnex 40

3.2.2.4. Risolutore dei vincoli ... 45
3.2.3. L’isolamento del codice ... 48
3.2.3.1. Tipi di oggetti fittizi ... 48
3.2.3.2. Microsoft Moles ... 49
3.2.3.3. Pex e Moles ... 51
3.2.4. Avvio di Pex ed utilizzo ... 55
3.2.4.1. Avvio di Pex da dentro l’ambiente di sviluppo Microsoft Visual Studio ... 55
3.2.4.2. Avvio da linea di comando ... 57
3.2.4.3. Reportistica ... 59
3.2.4.4. Documentazione e supporto a disposizione ... 60
CAPITOLO 4 Strumenti per la generazione ed esecuzione automatica di Test funzionali per applicazioni web ... 62
4.1. Strumenti esistenti ... 62
Selenium automates browsers ... 62
WatiN ... 63
Sahi ... 63
SoapUI ... 64
4.2. WatiN, libreria C# per l’automazione della navigazione di applicazioni web ... 64
4.2.1. Caratteristiche ... 64
4.2.2. L’utilizzo ... 65
4.2.2.1. Configurazioni per l'avvio ... 66
4.2.2.2. Le librerie ... 69
4.2.2.3. Classiche implementazioni ... 71
4.2.3. Documentazione ... 71
4.2.4. Conclusioni ... 72

4.3. Strumento sviluppato per la generazione dei dati di input per il Testing di pagine web utilizzando la libreria WatiN 73

4.3.1. Introduzione 73

4.3.2. Funzionalità 74

4.3.3. File di configurazione 77

4.3.4. Algoritmo di esecuzione 80

4.3.5. Note di sviluppo ed utilizzo 83

4.3.6. Sviluppi da ultimare 84

CAPITOLO 5 Analisi sperimentale dei casi di studio 86

5.1. White-box Testing 86

5.1.1. Microsoft Pex a confronto con i software di ricerca sui sorgenti Benchmarks Hand-Crafted 86

5.1.1.1. Sorgenti analizzati 86

5.1.1.2. Risultati e valutazioni di Pex 88

5.1.1.2.1. Descrizione delle esplorazioni effettuate 88

5.1.1.2.2. Risultati delle esplorazioni di Pex messe a confronto con quelle degli strumenti sperimentali analizzati 89

5.1.2. Microsoft Pex sul caso di studio di ETT s.r.l. 93

5.1.2.1. Applicazione Archivia Log 93

5.1.2.1.1. Spiegazione dell'utilizzo di Pex sul caso di studio Archivia Log 94

5.1.2.1.2. Risultati riscontrati delle esplorazioni sul caso di studio Archivia Log... 101

5.1.2.2. Libreria Ett.Common 103

5.1.2.2.1. Utilizzo di Microsoft Pex sul caso di studio della libreria Ett.Common . 103

5.1.2.2.2. Risultati riscontrati sul caso di studio della libreria Ett.Common 107

5.2. Black-box Testing sulle applicazioni web di ETT s.r.l. 109

5.2.1. Testing sul sito web COL attraverso lo strumento sviluppato 109

5.2.1.1. Risultati sulla generazione automatica dei Test del sito web COL 111

5.2.1.2. Risultati sull'esecuzione automatica dei Test del sito web COL 112

5.2.2. Testing sull’applicazione web ClicLavoro Campania attraverso lo strumento sviluppato 113

5.2.2.1. Risultati sulla generazione automatica dei Test del sito ClicLavoro 116

5.2.2.2. Risultati sull'esecuzione automatica dei Test del sito ClicLavoro 116

5.2.3. Testing sul sito web Portale Marche Multifunzionale attraverso Unit Test con l'utilizzo della libreria WatiN 117

5.2.3.1. Risultati sull'esecuzione delle Unit Test sul sito web Portale Marche Multifunzionale 118

CAPITOLO 6 Conclusioni 120

Bibliografia X

CAPITOLO 1 Introduzione

La diffusione delle applicazioni software dovuta all'espansione negli ultimi anni delle infrastrutture digitali ha portato ad un maggior controllo del suo corretto funzionamento.

Si pensi alle applicazioni web accessibili da Internet[1] da milioni di utenti finali, ai software di controllo di apparecchiature mediche dove un malfunzionamento potrebbe comportare un malessere ad una persona umana oppure ad un software dove un malfunzionamento comporterebbe una perdita di immagine dell'azienda produttrice.

Argomento di questa tesi è lo studio della verifica in modo automatizzato del corretto funzionamento di una qualsiasi applicazione, partendo dal suo sviluppo fino all'utilizzo dell'utente finale.

ETT s.r.l.[2] [1], azienda dove si è svolto gran parte del lavoro, ha permesso di affrontare questa tematica essendo, l'attività di testing, presente in qualsiasi progetto e considerando la tematica importante e di interesse.

1.1. Motivazioni

Partendo dall'idea che un prodotto viene acquistato se esso funziona, per esserne certi, è necessario effettuare delle verifiche su di esso attraverso prove di utilizzo chiamate *test* da cui deriva il nome dell'attività di *testing*. Si è anche dimostrato che spendendo più ore nella preparazione dei processi di testing alla fine del progetto si ha un risparmio in ore di mantenimento e correzione di anomalie.

Se un prodotto è virtuale come il software[3], non fisicamente riconoscibile come un prodotto fatto di materia, questo concetto diventa più complesso in quanto entra in gioco il concetto di duplicazione (lo stesso prodotto utilizzato in contesti diversi), dinamicità del prodotto finale e cambiamento di comportamento in base alla personalizzazione portando

[1] Internet : Dall'inglese "INTERconnected NETworks" letteralmente "reti interconnesse", di fatto la rete informatica mondiale di reti di computer con accesso pubblico.
[2] ETT s.r.l. : Electronic Technology Team, società di informatica genovese dove è stato svolto il tirocinio che ha permesso la scrittura di questo documento.
[3] Software : Più comunemente chiamato programma o applicazione.

facilmente alla perdita del controllo del prodotto in origine, la verifica diventa quindi più articolata e difficile da effettuare.

L'attività di testing è svolta a diversi livelli del ciclo di vita del software e si differisce per tipologie in base a cosa si vuole verificare. Due grandi distinzioni per la verifica del prodotto sono quella strutturale e quella funzionale. La prima verifica che ogni singola parte di cui è composto il prodotto funzioni nella sua unità, facendo l'analogia con una macchina si vorrebbe che quando si gira il volante, girino anche le ruote, se ciò accade, sia il volante che le ruote lavorano correttamente. La seconda, la verifica funzionale, accerta che il prodotto rispecchi le funzionalità per cui è stato creato. Utilizzando la metafora della macchina si vorrebbe che nel momento in cui si gira il volante a destra, anche le ruote e di conseguenza la macchina, girino a destra.

In qualsiasi ambito le verifiche vengono sempre effettuate prima della messa in produzione di un prodotto, le verifiche strutturali spesso vengono dimenticate poiché ciò a cui si da importanza è il risultato finale, quello funzionale. Si vuole quindi, oltre a studiare ed approfondire come effettuare tali verifiche e agevolare lo sviluppatore creatore dell'applicazione, cercare ed analizzare strumenti che automatizzano l'analisi strutturale dell'applicazione generando test per verificare la robustezza di ciò che si è sviluppato.

L'interesse di trovare dei buoni strumenti per effettuare test strutturali in modo automatico, è molto alto nella ricerca in quanto è difficile avere uno strumento efficiente che, con pochi test, analizzi e verifichi tutto il codice.

Le verifiche funzionali normalmente vengono eseguite manualmente, il Tester [4] conoscendo le funzionalità che deve avere l'applicazione, effettua delle prove e confronta i risultati con quelli aspettati. Negli ultimi anni con l'espansione della rete Internet, le applicazioni sono diventate distribuite, accessibili da diverse postazioni e basate su tecnologie web (letteralmente ragnatela)[5]. Per questo motivo ci si focalizza sullo studio dell'esecuzione di test funzionali esclusivamente su applicazioni web. Si vuole anche trovare un modo per automatizzare i test ed aiutare il tester nella generazione dei dati da

[4] Tester : Persona finisca che esegue delle prove sul sistema da testare.
[5] Web : Letteralmente “ragnatela” è diventata una parola comune nell’indicare i collegamenti tra diverse applicazioni funzionanti da remoto e su Internet o sviluppate attraverso tecnologie per poterlo fare.

utilizzare in modo da limitare il lavoro del tester alla sola verifica concettuale per capire se l'applicazione testata funziona correttamente oppure no.

In ETT s.r.l. le attività di testing sono svolte da un gruppo di persone predisposte per queste attività, esse si dimostrano sempre onerose in termini di risorse e tempo e non esistono automatismi.

1.2. Obiettivi

L'azienda ETT s.r.l. sviluppa principalmente applicazioni web in ambiente Microsoft .NET, la ricerca di strumenti si è quindi orientata in questo ambito.

Un primo obiettivo è la ricerca di strumenti per l'esecuzione di test per verifiche strutturali, mettendoli a confronto con altri già esistenti e utilizzarne almeno uno per i progetti in corso di sviluppo.

Un secondo obiettivo è quello di arrivare all'automazione dei test funzionali. L'idea è di cercare strumenti esistenti e si metterli a confronto. Si vuole inoltre cercare una libreria che permetta l'automazione della navigazione di applicazioni web per poter progettare uno strumento per eseguire test funzionali in modo semi-automatico, strumento da fornire al gruppo di tester per velocizzare i processi di testing e poter ripetere facilmente i test svolti e da svolgere.

La ricerca e l'analisi di strumenti di testing considera sia prodotti commerciali, che prodotti open source e sperimentali.

1.3. Organizzazione della Tesi

A seguito di questo capitolo introduttivo, si propone nel CAPITOLO 2 - Il Testing del software, un breve ripasso sull'argomento del testing trattato dalla scienza dell'ingegneria del software.

Si prosegue con il CAPITOLO 3 - Strumenti per la generazione automatica di Test strutturali, per affrontare la prima grande tematica dei test strutturali. Analizzando gli strumenti presenti sul mercato e presentando nel dettaglio lo strumento Microsoft Pex [2].

Nel CAPITOLO 4 - Strumenti per la generazione ed esecuzione automatica di Test funzionali per applicazioni web, analogamente al CAPITOLO 3, si tratterà la tematica dei test funzionali e si presenteranno gli strumenti ricercati. Anche in questo caso verrà presentata nel dettaglio la libreria di sviluppo WatiN [3] e verrà spiegato lo strumento sviluppato per l'automazione di questa tipologia di test.

Nel CAPITOLO 5 - Analisi sperimentale dei casi di studio, si riportano i risultati trovati sulla comparazione dello strumento Pex con alcuni strumenti sperimentali, si documenterà l'utilizzo di Pex in alcuni progetti di sviluppo di ETT s.r.l. ed infine verrà presentato l'utilizzo dello strumento sviluppato per il testing di alcuni siti web sempre sviluppati in ETT s.r.l.

Nell'ultimo CAPITOLO 6 - Conclusioni, si trarranno le conclusioni sul lavoro svolto.

CAPITOLO 2 Il Testing del software

L' Institute of Electrical and Electronics Engineers[6] definisce il testing come "*il processo di valutazione di un sistema attraverso strumenti manuali o automatici col fine di determinare se il sistema soddisfa i requisiti specificati oppure se il suo comportamento verificato differisce da quello atteso*".

A differenza di un comune prodotto non informatico, lo sviluppo di un software può dipendere da altri sistemi o da altri software, pertanto il testing può essere fatto più volte durante le fasi del ciclo di vita del software in base al modello di sviluppo adottato.

Al fine di installare in produzione un software finito, lo scopo del testing è quello di rilevare i difetti tramite i malfunzionamenti per poterli correggere prima che venga utilizzato dall'utente finale.

Teorema di Dijkstra (1969):

"*il test di un programma può rilevare la presenza di malfunzionamenti, ma mai dimostrarne l'assenza*" .

2.1. Accenni ai modelli di sviluppo

L'ingegneria del software definisce *modello di sviluppo* il modo in cui i processi, definiti per la progettazione e messa in esercizio di un programma software, interagiscono e si connettono tra di loro.

Le singole attività descritte nel ciclo di vita del software sono:

- Definizione dei requisiti ed analisi di progetto;
- Progettazione degli sviluppi;
- Implementazioni;
- Collaudo delle implementazioni;
- Rilascio;

[6] Institute of Electrical and Electronics Engineers, abbreviato IEEE, letteralmente "Istituto degli ingegneri elettrici ed elettronici" è una associazione internazionale per la standardizzazione delle tecnologie.

- Manutenzione.

A seconda del modello utilizzato, le attività possono essere ripetute più volte, affrontate ad alto livello con la stesura di documenti oppure ad un livello più basso con la sola fase di scrittura del codice. Alcuni modelli prevedono più incontri con il cliente e sviluppi incrementali, altri un incontro all'inizio ed uno alla fine prima della messa in esercizio.

L'evoluzione dei modelli è avvenuta in modo incrementale, il miglioramento del primo diventava un secondo modello. Primo fra tutti quello a "cascata" mentre negli ultimi anni si sta passando a modelli con metodologie dette "agili" .

In ordine cronologico, di seguito i più utilizzati:

- Modello a cascata;
- Modello evolutivo;
- Modello a spirale;
- Metodologie agili.

2.1.1. Descrizione dei modelli di sviluppo

Modello a cascata

Nato negli anni 70, è stato il primo ad introdurre il concetto di "fasi di sviluppo". In questo modello ad ogni errore si torna alla fase precedente o all'inizio. Già in questo modello c'era il concetto di testing, come penultima fase prima della messa in produzione.

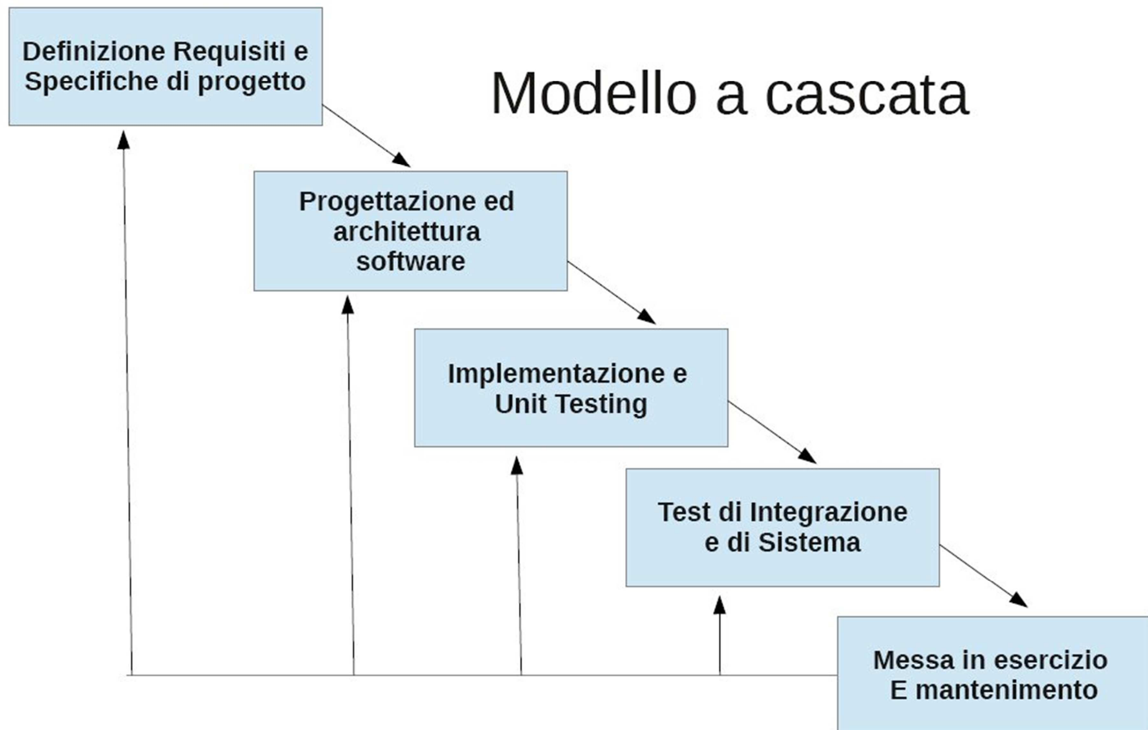


Figura 2.1: Modello di sviluppo a cascata

Modello evolutivo

Evoluzione del primo, aggiunge il concetto di "protopizzazione" creando software semi completi e procedendo per passi evolutivi (incrementali) fino ad arrivare ad avere un applicativo completo.

Modello a spirale

Arricchisce i modelli precedenti con il concetto di "rischio", ad ogni fase si analizzano quali potrebbero essere i rischi che si potrebbero correre seguendo la strada definita nella fase. Il collaudo viene fatto ad ogni fase evolutiva.

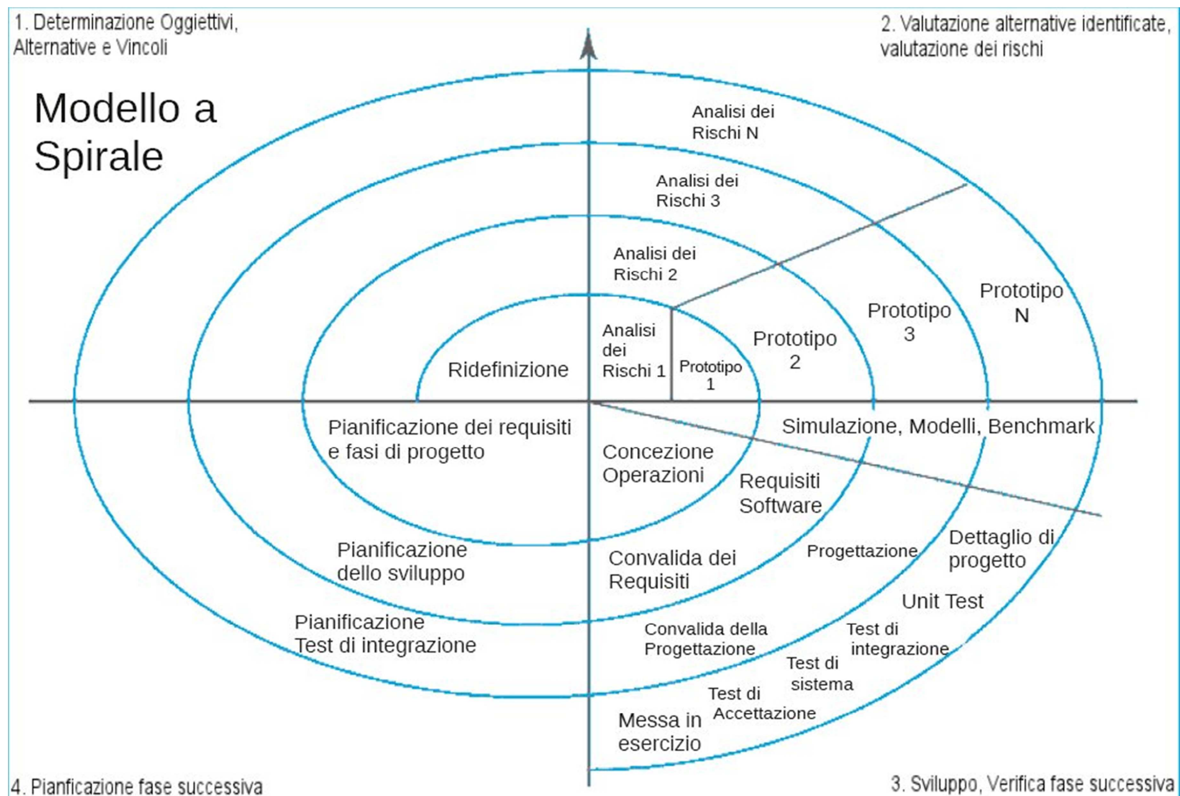


Figura 2.2 - Modello di sviluppo a spirale

Metodologie agili

Dagli anni '90 è nata una nuova filosofia di sviluppo, le attività definite nei modelli precedentemente creati dall'ingegneria del software sono considerate "pesanti", “lente” e non competitive con il mercato a fronte dell'evoluzione tecnologica in corso. Si è pensato di creare delle metodologie più "leggere" ed "agili" per affrontare i problemi più rapidamente, direttamente con il cliente, senza dover aspettare la fase successiva come accade nei modelli precedentemente descritti. L'ordine delle attività non è più sequenziale ma dinamico in base all'andamento dello sviluppo.

Un esempio di metodologia agile è l' "Extreme Programming" (letteralmente “programmazione estrema”, abbreviata in XP), un approccio all'ingegneria del software formulato da Kent Beck, Ward Cunningham e Ron Jeffries. Kent Beck, scrisse il primo libro sull'XP, Extreme Programming Explained: Embrace Change [4]. Aspetti interessanti sono la programmazione a più mani (generalmente in coppia), la verifica continua del programma durante lo sviluppo per mezzo di programmi di test e la frequente reingegnerizzazione del software a passi incrementali, senza dover rispettare fasi di

sviluppo particolari. Vengono adottati processi di sviluppo come la Test Driven Development (letteralmente “sviluppo guidato alle verifiche”, abbreviato in TDD), dove in base alle funzionalità e specifiche che si dovranno implementare, si creano prima i test e poi il codice sorgente.

Le fasi della TDD sono le seguenti :

1. Sviluppo del test;
2. Verifica del fallimento del test (non esiste ancora il codice che deve essere testato);
3. Sviluppo del codice per "passare" il test;
4. Verifica del superamento del test;
5. Reingegnerizzazione del software per l'integrazione con il resto del software.

Si noti che in qualsiasi modello di sviluppo che si adotti, la fase di testing sarà sempre un passo fondamentale per la buona riuscita del progetto.

2.2. Verifica e convalida

Occorre distinguere i "malfunzionamenti" del software ("failure") dai "difetti" derivati da errori nello sviluppo ("fault", "defect" o "bug").

Un malfunzionamento è un comportamento del software diverso dai requisiti di progetto e si presenta quando il sistema non esegue quello che l'utente si aspetta.

Un difetto è una parte di software o processo, che quando eseguito in particolari circostanze, genera un malfunzionamento.

Riassumendo :

- *Errore*: deriva da una causa umana ed è poco prevedibile;
- *Difetto*: è introdotto da un errore ed è da eliminare;
- *Malfunzionamento*: è prodotto da un difetto ed è da riparare;
- *Scostamento*: è la differenza tra l’osservato e il desiderato.

Le attività di controllo si differenziano in "verifica" e "convalida".

La verifica assicura che il sistema integri tutti i requisiti funzionali definiti in analisi di progetto. La convalida assicura che il software sia conforme con le aspettative dell'utente e che quindi le funzionalità siano effettivamente quelle richieste.

L'attività di verifica è particolare, si può ottenere un software perfettamente funzionante, senza errori, ma del tutto inutile in quanto non rispecchia quanto era stato richiesto all'inizio, aspetto evidenziato da una attività di convalida.

Le attività di verifica si dividono in due famiglie: "verifica statica" o "metodi formali" ed "analisi dinamica" da cui deriva il termine *Testing*.

Le attività di verifica vengono eseguite più volte durante lo sviluppo del software in base al modello di sviluppo utilizzato.

2.2.1. Verifica statica

Verifiche di questo tipo vengono fatte sui requisiti formulati, sul progetto e sul codice sorgente.

L'utilizzo di una verifica statica sul codice sorgente è possibile farla solo se il programma non dipende da altri, in questo tipo di verifica non si potrebbe controllare cosa comporterebbe tale dipendenza.

La verifica del codice sorgente senza la sua esecuzione può essere fatta manualmente attraverso la semplice lettura del codice sviluppato e basandosi sui documenti di progetto redatti.

La verifica viene fatta su uno o più componenti software in modo separato.

Esistono “metodi formali” per la verifica statica attraverso strumenti automatici di analisi del codice sorgente ed estrazione di misure calcolate per una analisi statistica del codice analizzato. Per esempio basandosi su teoremi (condizioni di verifica) la cui verità implica che il verificarsi di certe pre-condizioni assicura il verificarsi di determinate post-condizioni oppure verifica che i dati del programma restino entro i limiti del loro tipo di

dato e nella precisione desiderata preservando problemi di overflow [7] o errori di arrotondamento. Linguaggi evoluti assegnano limiti statici a tipi discreti consentendo verifiche automatiche sulle corrispondenti variabili.

2.2.2. Verifica dinamica

Si applica ai singoli componenti del sistema o al sistema nella sua interezza. Considerando il software in esecuzione, si interpretano le dinamiche dipendenti dai fattori esterni ed interni al sistema. Questa tipologia di verifica comporta l'utilizzo di numerose risorse e lunghi tempi per la preparazione degli ambienti di simulazione uguali all'ambiente di produzione.

Le attività di verifica dinamica prevedono :

1. Pianificazione della "prova di esecuzione" sulla base di casi di studio (test case) dove si andranno a definire :
    - I dati di input[8] del programma da eseguire;
    - Gli output[9] aspettati;
    - L’ambiente o configurazione del programma.
2. Esecuzione del caso di studio;
3. Analisi e valutazione dei risultati con eventuali successive attività correttive.

Durante il ciclo di vita del progetto di sviluppo si collezionano numerosi casi di studio, raccolti in una o più batterie detta *Test suite*. In una test suite normalmente si prevedono l'esecuzione di singoli test case in modo sequenziale .

Si definisce “procedura di test” il procedimento (automatico o manuale) per eseguire, registrare, analizzare e valutare i risultati delle prove eseguite.

2.3. Tipologie di Testing

[7] Overflow : Evento non gestito da un calcolatore che si verifica quando un’operazione (normalmente matematica) dà un risultato così elevato da non poter essere gestito dal software che ha eseguito l’operazione.
[8] Input: Letteralmente immettere, normalmente si indicano i dati in ingresso di qualcosa, un test o un programma.
[9] Output: Letteralmente emettere, normalmente si intende un risultato in uscita di una elaborazione.

In base all'obiettivo preposto ed il livello di software considerato nelle singole attività di testing, si differenziano le seguenti tipologie :

- Unit Testing;
- Test di Regressione;
- Test di Integrazione;
- Test di Sistema;
- Test di Accettazione (Alpha Testing, Beta Testing).

I modelli di sviluppo, in base alla fase del progetto corrente, utilizzano tipologie di testing differenti .

2.3.1. Unit Testing

Letteralmente "Unità di prova" anche già precedentemente chiamata *Test case*, è l'attività volta a determinare la correttezza e completezza, rispetto ai requisiti, di un programma visto come singolo modulo.

Riconosciuto come modello standardizzato da IEEE (1008-1987) [5], è composto dalle seguenti attività:

1. Pianificazione dell’approccio, risorse e tempistiche previste;
2. Individuazione delle caratteristiche da testare;
3. Determinazione dell’insieme di test;
4. Esecuzione dei test;
5. Verifica se altri test sono necessari;
6. Valutazione dei risultati.

Il concetto di unità o isolamento del modulo, comporta particolare attenzione per i riferimenti a servizi derivanti da componenti esterni. Essi vengono simulati attraverso l'uso di "Stubs" o "Driver" per "sostituire" o "pilotare" il flusso di controllo del singolo test inerente al contesto che si sta testando. Altre tecniche di simulazioni prevedono il "Mock" o "Fake" dei dati utilizzati per non accedere ai sistemi esterni e poter testare la sola unità di codice sorgente, simulando il valore del dato che normalmente è fornito dal sistema esterno.

2.3.2. Test di Regressione

Ad ogni aggiornamento di un modulo software la nuova versione deve mantenere le funzionalità di quella precedente. Un test di regressione considera il caso di verifica di questa compatibilità attraverso l'esecuzione dei due programmi (vecchio e nuovo) sugli stessi dati (eventualmente convertiti nel nuovo formato) ed il successivo confronto dei risultati.

I test di regressione possono essere fatti a livello di singolo modulo o sull'intera test suite.

2.3.3. Test di Integrazione

Questa tipologia di test, verifica la correttezza funzionale nell'iterazione tra più moduli.

Di seguito diverse tecniche per testare l'integrazione tra i moduli:

- Assemblare i moduli tra loro in modo incrementale;
- Assemblare produttori prima dei consumatori dove la verifica dei primi fornisce ai secondi un flusso di controllo (chiamate) e flusso dei dati già corretti;
- Assemblare tutti i moduli e testare il sistema come unità (anche detto "Big Bang Test", letteralmente "prova a grande scoppio").

Ogni singolo test di integrazione normalmente fa riferimento ad un flusso operativo dell'applicazione.

2.3.4. Test di Sistema

Per sistema si intende l'applicativo software completo, messo in esercizio ed arrivato all'obiettivo del progetto. Il test di sistema è volto a testare particolari proprietà globali di esso. Si possono quindi distinguere :

- *Test di stress*: per verificare le proprietà del sistema in condizioni di sovraccarico;
- *Test di robustezza*: per verificare le proprietà del sistema quando i dati trattati non sono corretti;
- *Test di sicurezza*: per verificare le proprietà di sicurezza del sistema.

2.3.5. Test di Accettazione

Fanno parte dell'ultima fase prima della messa in esercizio.

Sulla base di dati forniti dall'utente, viene testata una speciale versione del software chiamata "Alpha Testing"[10]. Successivamente è possibile rilasciare in produzione oppure utilizzare una seconda metodologia, che potrà essere utilizzata ad ogni rilascio. Essa consiste nella messa in produzione di una versione del programma chiamata "Beta Testing", disponibile per pochi utenti finali che svolgeranno l'attività di accettazione usando il sistema ad un prezzo vantaggioso per riportarne gli eventuali problemi. In questo secondo caso dopo le correzioni segnalate, la nuova versione verrà rilasciata a tutti gli utenti finali. Questa seconda metodologia si adotta quando la mole di utenti finale è alta ed il sistema è molto complesso.

2.4. Criteri e determinazione dell'insieme di Test

La determinazione dei dati di input di un test ne determina l'efficacia. Nell'insieme una Test suite "ottima" porterà ad un buon risultato del prodotto finale, ma la determinazione di questi dati non è semplice, come evidenziato dal teorema seguente.

Teorema di Howden (1975):

"*non esiste un algoritmo che, dato un programma P qualsiasi, generi per esso un test finito ideale (definito da un criterio affidabile e valido)*".

La creazione di un test "ideale" per un programma *P* non è un problema "banale".

2.4.1. Criteri di selezione

Di seguito alcune definizioni per capire come l'ingegneria del software ha affrontato il problema per la creazione di un test "ideale".

Teorema:

[10] Alpha Testing: Alpha è la prima lettera dell'alfabeto greco, idealmente quindi la primissima versione funzionante da verificare. Potranno seguire ulteriori versioni, Beta (seconda lettera) e così via.

"*Un programma P è una funzione derivante da un insieme di dati D (Dominio) in un insieme di dati R (Codominio)*"

$$P: D \subset R$$

Formula 2.1

Dato :

$d \in D$

Tesi:

P(d) è corretto (ok(P, d)) se soddisfa le specifiche, non corretto altrimenti.

Teorema:

"*Un test T per P è un sottoinsieme dei dati D*"

Tesi:

P è corretto in T (ok(P, T)) se per ogni t $\in$T, si ha ok(P, t).

P è corretto (ok(P)) se P è corretto in D.

Un test T è ideale se la correttezza di P in T implica la correttezza di P. (da questo segue che il test D è ideale).

Teorema:

"Un criterio di selezione C per P è un insieme di sottoinsiemi dei dati D"

Tesi :

Un criterio di selezione C è affidabile se comunque presi T1 e T2 selezionati da C, si ha

$$ok(P, T1) \Leftrightarrow ok(P, T2)$$

Formula 2.2

Un criterio di selezione C è valido se

$$\exists t\ (\text{selezionato}(t, C)\ \wedge\ \neg \text{ok}(P, t))\ \neg \text{ok}(P)$$

Formula 2.3

Teorema (Goodenough, Gerhart):

"Il fallimento di un test T per un programma P selezionato da un criterio C affidabile permette di dedurre la correttezza del programma P"

Tesi:

$$\text{affidabile}\ (\ C\ , P\ )\ \wedge\ \text{valido}\ (\ C\ , P\ )\ \wedge\ \text{selezionato}\ (\ C\ , T\ )\ \wedge\ \neg\ \text{successo}\ (\ T\ , P\ )\ \Rightarrow\ \text{ok}(\ P\ )$$

Formula 2.4

Una singola prova non basta, i suoi risultati basterebbero solo per verificare la sua singola esecuzione. Le prove non possono essere generalizzate e devono essere ripetibili. Esse sono costose, richiedono molte risorse (tempo, persone, infrastrutture), e quindi necessitano di un processo definito per le attività di ricerca dei malfunzionamenti, analisi degli errori e correzione degli stessi.

2.4.2. Criteri fondamentali

Esistono due tipologie di criteri di selezione dei test

- *Test funzionale o Black-Box Testing*[11]
    - Basato sulle specifiche del programma e sulla conoscenza delle sole funzionalità;
    - Ricerca l'affidabilità e l'efficienza;
- *Test strutturale o White-Box Testing*[12]
    - Basato sul programma con la conoscenza del codice;
    - Indipendente dalle specifiche;
    - Con il solo obiettivo della ricerca dei difetti.

[11] Black-Box : letteralmente "scatola nera" dovuto al fatto che non si vuole conoscere la struttura del codice sorgente del sistema

[12] White-Box : letteralmente "scatola aperta" dovuto al fatto che si ha piena conoscenza del codice sorgente del sistema

Questi due criteri sono complementari ed ognuno può rilevare malfunzionamenti non rivelabili con l'altro.

2.4.2.1. Black-Box Testing

Di seguito due tecniche per la generazione dei dati per effettuare le verifiche funzionali.

Partition Testing

Il partition testing (letteralmente, "partizionamento di prove") è l'approccio più caratteristico del testing funzionale, l'idea è quella di suddividere in sottoinsiemi di dati gli input da testare, da cui deriva il nome partizionamento.

Ogni partizione deve essere disgiunta dalle altre ed ogni valore deve avere una caratteristica in comune con gli altri valori della partizione. Tutti i valori devono essere nel dominio *D* dei dati di ingresso del programma. Tali valori potrebbe essere infiniti.

Nella procedura di testing si possono eseguire i seguenti passi :

1. Definire le caratteristiche dei dati in base alle funzionalità da testare;
2. Partizionare per caratteristiche;
3. Combinare i valori per caratteristiche creando i dati di input da utilizzare nei test, per verificare il corretto funzionamento ad ogni partizione dei dati.

Una parte fondamentale di questa procedura è definire un modello della rappresentazione dei dati di input più comunemente chiamato "Input Domain Model",(letteralmente "modello del dominio degli input", abbreviato IDM). Identificando le funzioni controllabili ed i moduli dell'applicazione da testare, tutti i parametri dei metodi e i campi dei moduli da inserire, si possono ricavare i dati per eseguire i test. Applicando un criterio di test per la generazione dei valori, sarà necessario scegliere una combinazione di essi, in base alle funzionalità.

Per definire IDM, esistono due approcci :

- *Basandosi sull'interfaccia che si sta testando*: i valori vengono generati identificando la tipologia di ogni singolo dato. Per esempio generando un valore numerico, un testo se è una stringa oppure una data;

- *Basandosi sulle funzionalità*: avendo una visione completa del comportamento del programma si possono generare valori per specifiche funzionalità. Più difficile da progettare e sviluppare ma con migliori risultati rispetto all'approccio precedente.

Nella scelta dei valori di ingresso, verificando sempre la completezza nell'insieme delle casistiche e ricordandosi di mantenere la disgiunzione tra le singole partizioni, si può far riferimento a diverse strategie:

- Includere valori validi, non validi e speciali;
- Includere valori ai confini dei domini, valori al limite dell'intervallo dei valori possibili. Questa strategia definisce un nuovo criterio di selezione conosciuto come "Boundary value analysis" (letteralmente "analisi ai confini dei valori");
- Includere valori che rappresentano un "uso normale" del sistema;
- Includere un numero considerevole di valori con le stesse caratteristiche per effettuare test di carico.

La combinazione dei blocchi di valori definiti, non è un problema banale, esistono un insieme di criteri denominato "Input Space Partitioning" (letteralmente "spazio dei dati di input partizionato", abbreviato ISP) per definire le possibili combinazioni, di seguito i criteri più comuni.

All Combinations

Letteralmente "tutte le combinazioni" è il criterio più semplice con la scelta di tutte le possibili combinazioni, tutti i valori di un blocco combinati con tutti i valori degli altri. E' anche il criterio più oneroso in termini numero di test generati AC.

Tesi:

AC è dato dal prodotto delle lunghezze dei blocchi di valori

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

$$AC = \prod (B_i)_{i=1}^{Q}$$

Formula 2.5

Each Choice

Letteralmente "ogni scelta", definisce che un valore per ogni blocco di caratteristiche dovrà essere usato in almeno un caso di test. Il numero di test massimo generato è EC .

Tesi:

EC è dato dalla lunghezza massima di valori dei blocchi utilizzati

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

$$EC = Max(B_i)_{i=1}^{Q}$$

Formula 2.6

Pair-Wise

Letteralmente "coppia di caratteristiche", considerando un valore per ogni blocco per ogni caratteristica, esso si combina con un valore di ogni blocco di un'altra caratteristica.

Tesi:

Il numero di test PW, sarà almeno il prodotto dei due più grandi blocchi di caratteristiche

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

B_j: il numero di valori del blocco j

$$PW = Max(B_i)_{i=1}^{Q} * Max(B_j)_{j=1,j!=i}^{Q}$$

Formula 2.7

t-Wise

Letteralmente "gruppo t di caratteristiche", è l'estensione della precedente combinazione *Pair-Wise,* definisce la scelta delle combinazioni di un gruppo di valori anziché solo due e di un valore per ogni blocco di ogni gruppo di caratteristiche, combinato con gli altri.

Tesi:

Il numero di test tW, sarà almeno il prodotto dei più grandi blocchi di caratteristiche considerati

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

t: il numero di blocchi raggruppati

$$tW = (Max(B_i)_{i=1}^{Q})^t$$

Formula 2.8

$$t = Q \implies tW = AC$$

Formula 2.9

Base-Choice

Letteralmente "scelta di base", si fonda sul principio che il tester riconosce certi valori come valori più importanti di altri, viene quindi scelto un valore di un blocco importante combinato con i valori negli altri blocchi.

Tesi:

Il numero di test BC, sarà il test di base sommato ad un test per ogni altro blocco

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

$$BC = 1 + \Sigma(B_i - 1)_{i=1}^{Q}$$

Formula 2.10

Multiple Base-Choice

Letteralmente "scelta di base multipla", evoluzione della precedente combinazione *Base-Choise,* anziché un valore in un blocco, esso viene scelto in più blocchi importanti e combinato con gli altri.

Tesi:

Il numero di test MBC, sarà dato dal numero di test di base scelti, sommato ai test effettuati degli altri blocchi

Dato:

Q: il numero di blocchi di valori del test da creare

B_i: il numero di valori del blocco i

M : il numero di blocchi di base ritenuti importanti

m_i: il numero di valori del blocco di base i

$$MBC = M + \Sigma(\mathrm{M} * (B_i - m_i))_{i=1}^{Q}$$

Formula 2.11

$$M = 1\,, \qquad m_i = 1 \implies MBC = BC$$

Formula 2.12

Error based Testing

Letteralmente "prova basata sugli errori", questo criterio è pensato per evidenziare specifici errori o classi di errori, partendo dalle funzionalità si genera un insieme di dati di input per verificare che l'esatto contrario porti ad un errore.

2.4.2.2. White-Box Testing

Questo criterio si basa sul principio di analizzare il codice sorgente e basarsi su di esso per creare criteri di selezione (C) per i test.

Ad ogni programma si può associare un grafo di controllo.

Ogni nodo del grafo è un "comando" che può essere collegato ad altri nodi attraverso archi. Ogni nodo può avere archi in ingresso e in uscita. Un comando può essere una semplice istruzione oppure una condizione. Un comando di tipo condizione può avere più archi in uscita, un comando di tipo istruzione no. Un cammino comprende un insieme di nodi attraversati da una singola esecuzione del programma.

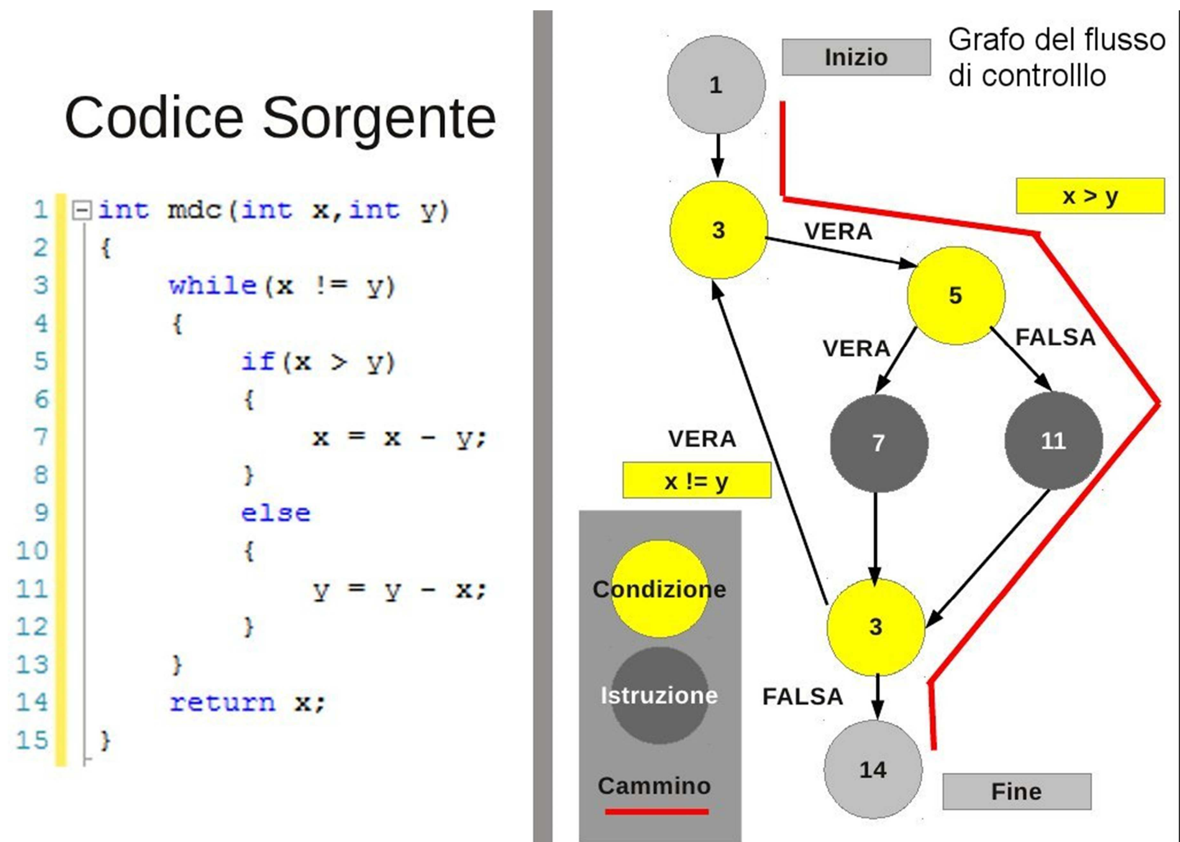


Figura 2.3: Sulla sinistra il codice sorgente di una funzione di calcolo, sulla destra il rispettivo grafo del flusso di controllo

Il White-Box Testing è basato sulla nozione di copertura, sull’esecuzione degli elementi del grafo del flusso di controllo di un programma.

Teorema:

Ad ogni test T basato sulla nozione di copertura è associabile una misura del grado di copertura dato

Tesi:

$$Grado\ di\ copertura\ = \frac{numero\ di\ elementi\ coperti\ da\ T}{numero\ totale\ di\ elementi\ copribili}$$

Formula 2.13

Si distinguono i seguenti criteri di selezione:

- Di copertura dei comandi;
- Di copertura delle decisioni;
- Di copertura delle condizioni;
- Di copertura delle decisioni e delle condizioni;
- Di copertura dei cammini.

È da precisare che questi criteri non forniscono un metodo per determinare effettivamente i dati di prova che causano l'esecuzione dei cammini selezionati, come evidenziato dal seguente teorema.

Teorema di Weyuker (1979):

Dato :

Un generico programma P

Tesi:

l'esistenza di un dato di test tale da causare l'esecuzione

- *di una particolare istruzione di P, oppure*
- *di una particolare condizione di P, oppure*
- *di un particolare cammino di P, oppure*
- *di ogni istruzione di P, oppure*
- *di ogni condizione di P, oppure*
- *di ogni cammino di P*

non è decidibile.

Per superare questo problema si utilizza la tecnica di "esecuzione simbolica".

Questa tecnica permette di determinare le condizioni che devono essere verificate dai dati in ingresso affinché un particolare cammino venga eseguito e relaziona i valori prodotti dall'esecuzione del programma con i valori in ingresso del programma stesso. Si noti che il programma non è eseguito con i valori effettivi ma con valori simbolici derivati dai dati di input. Dai valori simbolici vengono elaborate formule risolte in base all'esecuzione del programma.

Un esempio di sessione di test con esecuzione simbolica potrebbe essere il seguente:

1. Selezione del criterio di copertura desiderato;
2. Selezione di un insieme di cammini la cui percorrenza permette di soddisfare il criterio scelto;
3. Esecuzione simbolica dei cammini selezionati per determinare la condizione di percorrenza di ciascun cammino selezionato (per ogni cammino verranno create un insieme di formule);
4. Selezione di un dato di test per ogni condizione selezionata al passo 3. L'insieme dei valori così selezionati rappresenta un test per il programma in base al criterio di selezione scelto.

Algoritmo 2.1

Nel CAPITOLO 3 - Strumenti per la generazione automatica di Test strutturali, verrà analizzata questa tecnica sfruttata da strumenti software che analizzando i sorgenti di un programma ed applicando una "esecuzione simbolica" essi riusciranno a creare test automatici e parametrizzabili fornendo test case per arrivare al massimo grado di copertura del codice possibile. In particolare facendo riferimento ai passi definiti nell'Algoritmo 2.1, gli strumenti risolveranno le formule definite nel passo 3 ed elaboreranno i dati di input da utilizzare nel passo 4.

CAPITOLO 3 Strumenti per la generazione automatica di Test strutturali

La verifica strutturale è basata sul concetto di copertura del codice, la sua mancanza indica chiaramente un rischio nell'esecuzione. Di seguito una analisi di strumenti software in aiuto allo sviluppatore per poter effettuare questa verifica.

3.1. Strumenti sperimentali e commerciali

Effettuando una ricerca per poter capire quali strumenti software esistono ad oggi per analizzare del codice scritto in C# [13], è emerso che vi sono pochissimi prodotti commerciali, qualche prodotto "open source" [14] utilizzabile a scopo non commerciale e qualche prodotto sperimentale.

3.1.1. Prodotti commerciali e open source

Questi prodotti sono specifici per le aziende che sviluppano e vendono software. È necessario verificare sempre il tipo di licenza in modo da capire se ciò che viene prodotto attraverso questi strumenti è permesso rivenderlo.

NConer Desktop

Rilasciato [6] come estensione di Visual Studio 2008 e 2010, è arrivato alla versione 4.0.

Funzionalità:

- Supporta i framework 2.0, 3.0, 3.5 e 4.0;
- Permette di analizzare il codice sorgente in modo semplice e veloce;

[13] C# : Linguaggio di programmazione ad oggetti ideato da Microsoft.
[14] Open source: Letteralmente "codice sorgente aperto". Indica un software i cui autori (più precisamente i detentori dei diritti) permettono e favoriscono il libero studio e l'apporto di modifiche da parte di altri programmatori indipendenti.

- Effettua la copertura di codice e analisi sintattica utilizzando classiche metriche di misura, estraendo: il numero di blocchi, namespace[15], classi, linee di codice , metodi, la massima profondità del grafo del flusso di controllo ed infine misura la complessità ciclomatica[16];
- Produce un report evidenziando i rami ed il grafo visitato;
- Produce un report con il dettaglio del codice sorgente evidenziato con colori diversi in base alla copertura effettuata esportabile in XML.

La versione per uso commerciale costa circa 660 $ (dollari americani) al momento della stesura di questo documento. Esiste anche una versione open source [7] che risulta molto limitata rispetto a quella commerciale e non è possibile utilizzarla se si vuole rivendere il codice analizzato.

C# Test Coverage Tool

Lo strumento [8] fa parte di una famiglia di prodotti orientati alla copertura di codice per diversi linguaggi. Installata l'applicazione, si può analizzare una libreria DLL (“Dynamic Link Library”, letteralmente “libreria a collegamento dinamico”) oppure un programma binario.

Funzionalità:

- Supporta i framework 2.0, 3.0, 3.5 e 4.0;
- Indipendente dalla piattaforma (funziona con gli strumenti Microsoft e Mono[17]);
- Durante l'analisi ha un basso consumo di risorse;
- Incrementa i casi di test che devono essere rieseguiti in base alle modifiche fatte nel codice a livello di metodo;
- Produce un report di copertura.

Per l’uso commerciale è necessario contattare il fornitore per poter avere un preventivo sul costo dello strumento.

[15] Namespace : Letteralmente “spazio di nomi” , indica il nome della libreria in cui una classe C# è stata dichiarata.
[16] Complessità ciclomatica: è una metrica software che consente di sapere quanto sia complesso il programma misurando il numero di percorsi linearmente indipendenti tramite il codice sorgente del software. Analizzando il risultato si potranno semplificare le parti più complesse.
[17] Mono: Piattaforma ideata per poter utilizzare le applicazioni .NET in sistemi operativi diversi da Microsoft Windows .

JetBrains dotCover

Rilasciato [9] come estensione di Visual Studio 2005, 2008, 2010 e 2012, è arrivato alla versione 2.2 . È anche stato incluso in altri strumenti come *ReSharper , dotTrace e dotPeek* più completi forniti dallo stesso produttore.

Funzionalità:

- Supporta anche applicazioni sviluppate con Silverlight [10];
- In base alla suite di test creata, evidenza il codice coperto e scoperto;
- Si può determinare quali test coprono una posizione particolare nel codice;
- Supporto per diversi framework di Unit Test: NUnit [11], xUnit [12] ,MSTest ed MSpec [13][18];
- Generazione del report di copertura del codice in diversi formati come XML, HTML o JSON[19].

Esistono diverse licenze. Per l'uso commerciale si possono spendere massimo 200 € (Euro), tasse escluse. È anche disponibile una versione studenti ed una versione open source gratuite.

Microsoft Pex

Pex ("Program Exploration", letteralmente "esploratore di programma") [2] è uno strumento software sviluppato dal gruppo di ricerca di Microsoft che esplora programmi sviluppati in C# e genera una suite di test per la verifica strutturale del codice. Verrà trattato nel dettaglio nel paragrafo 3.2 - Microsoft Pex, strumento di analisi per la creazione di Test automatici.

3.1.2. Strumenti sperimentali

[18] MSTest, NUnit, xUnit e MSpec: Sono librerie scritte per sviluppare unit test in ambiente .NET. È possibile definire una classe di test, un metodo di test, la configurazione iniziale e gli Assert (letteralmente affermazione) per definire se un test è passato o fallito. NUnit è molto conosciuto nel mondo dello sviluppo, xUnit è la versione open source di Microsoft mente MSTest è la versione integrata in Microsoft Visual Studio dalla licenza Premium. Infine MSpec è un prodotto di nicchia open source poco conosciuto.

[19] JSON : JavaScript Object Notation, nato come oggetto per la rappresentazione di dati nel linguaggio JavaScript è diventato una standardizzazione per la serializzazione di dati in diversi linguaggi di programmazione..

Volendo confrontare strumenti commerciali con strumenti sperimentali, di seguito una breve descrizione di uno strumento nato dalla ricerca americana ed un insieme di algoritmi sviluppati dal gruppo di ricerca dall'Università degli Studi di Genova del dipartimento STAR-LAB ("Systems and Technologies for Automated Reasoning Laboratory", letteralmente "Laboratorio di sistemi e tecnologie per il ragionamento automatico") [14], gruppo composto dal Prof. Enrico Giunchiglia , Prof. Massimo Narizzano, Dott. Emanuele Di Rosa e Dott. Gabriele Palma.

CHESS

Letteralmente scacchi [15] ma il suo scopo è quello di effettuare l'esplorazione del codice di programmi avviati in multithreading verificando la copertura del codice in situazioni di iterazione tra i singoli thread[20]. Sviluppato dal Microsoft Research non è stato possibile trovare l'applicativo e farne ulteriori analisi.

3.1.2.1. Software sperimentali sviluppati in ambito di ricerca

Sono stati analizzati quattro strumenti sperimentali allo scopo di generare test automatici per arrivare alla massima copertura del codice sorgente analizzato.

Tre di essi sono stati sviluppati dal gruppo di ricerca dell'Università degli studi di Genova, mentre per un quarto, studiato dal gruppo, sono stati analizzati i soli risultati trovati dalla sua esecuzione in laboratorio, confrontandoli con gli altri.

Ogni strumento rappresenta, attraverso i metodi formarli, il grafo del flusso di controllo come insieme di vincoli da risolvere per l'estrazione dei dati di input per la generazione dei test.

Un problema comune delle suite di test generate per effettuare test strutturali, è la ridondanza di un sottoinsieme di test dovuta alla normale complessità dell'analisi fatta per ricavarli, questi test ridondanti comportano l'aumentano dei costi di esecuzione.

[20] Thread : Letteralmente fili, nell'informatica si intende lo stesso programma avviato contemporaneamente più volte, utilizzando risorse diverse o comuni.

Di seguito una breve spiegazione sul funzionamento dei quattro strumenti, la spiegazione dettagliata è possibile leggerla nel documento “Automatic Generation of High Quality Test cases and Plans in Software Testing and Automated Planning” [16], letteralmente “Generazione automatica di test di alta qualità e della loro esecuzione automatica”.

TeGeVe

Questo strumento (“Test Generator for software Verification”, letteralmente “Generatore di test per la verifica del software”) [17] si basa esclusivamente su tecniche euristiche di analisi del codice utilizzando CBMC[21]. L'analisi è di tipo statica attraverso l’esecuzione simbolica. L'algoritmo di esecuzione è il seguente:

1. Viene definito un nodo da raggiungere;
2. CBMC genera un percorso e verifica se si raggiunge il nodo definito;
3. Se lo si raggiunge si definisce che il percorso è *feasible* (letteralmente fattibile) ed attraverso un SAT Solver, viene risolto il sistema di vincoli definito dal percorso generando gli input per la generazione del test;
4. Se non è feasible si cerca un altro percorso;
5. L'algoritmo si ferma quando sono stati raggiunti tutti i nodi raggiungibili.

Un problema di questo strumento è la difficoltà di esecuzione nel caso di un sorgente con numerosi percorsi non fattibili.

noPref

Con un approccio diverso da TeGeVe, vengono ricercati solo percorsi fattibili per raggiungere un nodo prescelto. L'analisi è di tipo statica attraverso l’esecuzione simbolica, di seguito l'algoritmo implementato:

1. Si definisce un nodo da raggiungere;
2. CBMC estrae un percorso feasible per raggiungerlo;
3. Attraverso un SAT Solver risolve il sistema dei vincoli definito dal percorso generando gli input per la generazione del test;
4. Si passa ad un percorso alternativo per raggiungere i nodi mancanti da coprire;

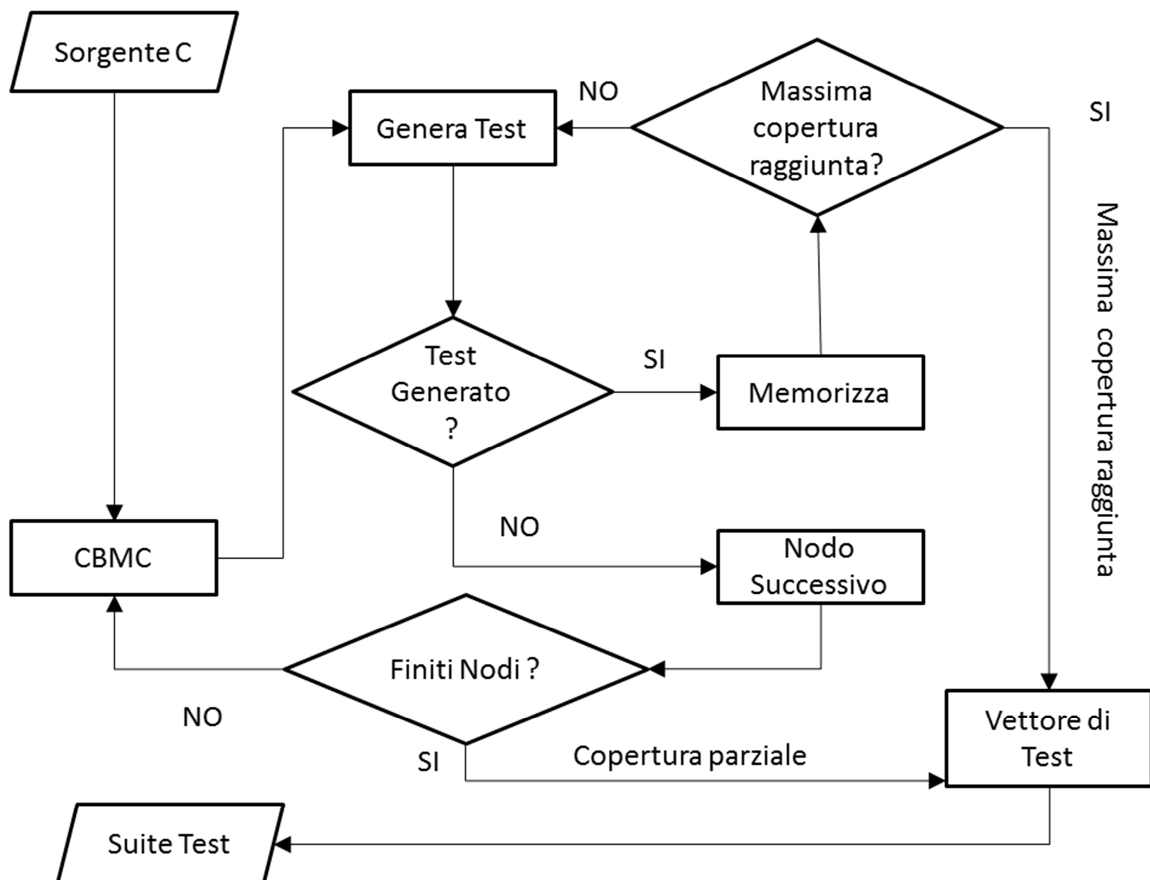
[21] CBMC : Bounded Model Checker for ANSI-C and C++, modello formale per l’analisi di grafi di controllo per l’analisi statica del codice sorgente

5. L'algoritmo si ferma quando sono stati raggiunti tutti i nodi scelti.

Un problema di questo algoritmo è nel passo della scelta del percorso successivo, non si utilizzano vincoli di ottimizzazione (non si fanno preferenze, da cui deriva il nome) e spesso vengono calcolati percorsi già parzialmente eseguiti.

SAT&PREF

[18] Evoluzione dello strumento noPref, rispetto ad esso, nel momento in cui si cerca un percorso successivo, vengono aggiunti vincoli in base ai nodi nel percorso ancora da visitare effettuando quindi delle preferenze.



Algoritmo 3.1- Funzionamento SAT&PREF

SAGE

Unico strumento [19] non sviluppato direttamente da gruppo STAR-LAB, effettua una analisi con l'esecuzione simbolica dinamica.

Analizzando un percorso, ricava il percorso successivo con la negazione dell'ultimo vincolo del percorso. Così facendo arriverà alla fine dell'analisi nel momento in cui saranno stati negati tutti i vincoli.

3.2. Microsoft Pex, strumento di analisi per la creazione di Test automatici

Tra i vari strumenti commerciali si è scelta l'analisi nel dettaglio di Microsoft Pex, in quanto strumento software sviluppato dal gruppo di ricerca di Microsoft e ritenuto interessante da un punto di vista funzionale.

L'idea di Pex è quella di generare una *Unit Test* per ogni metodo pubblico trovato nell'esplorazione in modo automatico. Eseguendo le unit test con parametri differenti, arriva a coprire tutto il codice possibile richiamato dal singolo test. Creando una serie di unit test parametrizzate, chiamate "Parameterized Unit Test" (letteralmente "unità di test parametrizzata", abbreviata PUT), Pex arriva alla massima copertura del codice possibile.

Ogni PUT è composta:

- Dai parametri con i valori assegnati durante l'esplorazione;
- Le assunzioni sui parametri perché essi abbiano dei valori validi;
- La chiamata al metodo da testare;
- L'accertamento o assert (letteralmente affermazione) del risultato (l'oracolo che certifica che il metodo chiamato si è comportato correttamente).

Una indagine di Microsoft nel 2010 ha indicato che il 79% dei sui sviluppatori utilizzano le unit test per documentare anche le esigenze del cliente, in modo da far riflettere le decisioni di progettazione sui test e proteggerli seguendo le modifiche da apportare [20]. Da qui l'idea di investire su uno strumento per aiutarli nel creare test in modo automatico.

Pex è abbinato ad un secondo framework, Microsoft Moles [21], che consente di isolare le unit test da componenti esterni richiamando metodi delegati reimplementati secondo le esigenze.

Effettuata l'esplorazione, Pex permette il salvataggio della suite di test generata per poter eseguire gli stessi test dopo un aggiornamento del programma (test di regressione).

È possibile configurare Pex per fargli creare le unit test utilizzando i framework più comuni come NUnit, xUnit ed ovviamente MSTest.

In questo capitolo, sulla base di questi concetti e sulla teoria dell'ingegneria del software evidenziati trattati nel CAPITOLO 2 - Il Testing del software, una breve analisi sul funzionamento di questo strumento.

3.2.1. Struttura applicativa

Pex è un componente aggiuntivo di Visual Studio ma è anche possibile utilizzarlo come strumento a linea di comando.

Nato nel 2008 ed arrivato alla versione 0.94.51023.0 è disponibile come estensione per le versioni di Visual Studio 2008 e 2010, per i framework .NET dalla versione 2.0 al 4.0 .

Scaricabile gratuitamente dal sito web ufficiale [2] è disponibile sotto forma di tre tipi di pacchetti :

- Pex e Moles versione Academic (letteralmente accademica, completamente gratuita) con licenza "Microsoft Research";
- Solo Moles versione "Visual Studio Gallery" (gratuita) con licenza "Visual Studio 2010 Power Tools";
- Pex e Moles per abbonati MSDN [22] con licenza "Visual Studio 2010 Power Tools" (per uso commerciale).

Essendo nato come progetto di ricerca si è mantenuta una versione di studio accademica per usi non commerciali.

Per Microsoft Moles è disponibile una versione separata per uso commerciale senza la necessità di un abbonamento MSDN.

Durante la scrittura di questo documento, mentre Moles nel framework 4.5 ha cambiato nome e si è evoluto nel framework Fakes [23], di Pex non esiste ancora una versione che supporti Fakes ed il nuovo framework.

3.2.2. Funzionamento dell'analisi effettuata sul codice da Pex

Avviata l'analisi dei sorgenti a disposizione, di una libreria DLL oppure di un file eseguibile binario, Pex utilizza la tecnica di esecuzione simbolica dinamica (abbreviata DSE, “Dynamic Simbolic Execution”) per poter navigare nel grafo del flusso di controllo del programma. La tecnica DSE è una variante della tecnica chiamata “esecuzione simbolica”.

Lo strumento esegue il codice più volte ed impara a conoscere il comportamento del programma monitorando le variazioni del flusso di controllo e dei dati ad ogni esecuzione diversa. In particolare ad ogni iterazione, esegue un nuovo ramo del programma non visitato in precedenza, costruendo un sistema di vincoli (uno per ogni nodo decisionale). Finita l'esecuzione Pex utilizza un solutore di vincoli chiamato Microsoft Z3 [24] per determinare i nuovi ingressi di prova da utilizzare nella prova successiva, attraverso la negazione dei vincoli non ancora negati e risolvendo il nuovo sistema creatosi. Il processo continua fino a quando non vi saranno più vincoli da risolvere.

La prima esplorazione viene fatta utilizzando dei valori di default sulla base del tipo di dato dei parametri da utilizzare.

3.2.2.1. Esempio di esplorazione

Ipotizzando di voler testare la seguente funzione scritta in C# :

```
1  public void Bar(int i, int j)
2  {
3      if (i < 0)
4      {
5          if (j == 123)
6              Console.WriteLine("Linea 6");
7      }
8      else
9          Console.WriteLine("Linea 9");
10 }
```

Esempio 3.1 - Funzione scritta in C# per analizzare l’esplorazione eseguita da Pex, a fianco in grigio il numero di riga del codice

Il comportamento aspettato per avere la copertura al 100% sarà di avere alla fine delle esecuzioni le due scritte in output[22] "$Linea$ 6" e "$Linea$ 9".

Pex si comporta nel seguente modo:

1. Effettua un primo test ($Test$ 1), utilizzando i valori di default, per il tipo di dati "int" utilizza "0"

   Bar(i=0,j=0)

   Nell'esecuzione di $Test$ 1 Pex:

   1.1. Esplora la riga 3 e 9 scrivendo nello standard output "$Linea$ 9";

   1.2. Ricava i seguenti due vincoli:

   $$\text{return target} != (\text{ClassMethod})\text{null};$$

   Equazione 3.1

   $$\text{return target} != (\text{ClassMethod})\text{null} \; \&\& \; -1 < \text{i};$$

   Equazione 3.2

   Dove nell'Equazione 3.1 la classe $ClassMethod$ contenente la funzione che si sta testando non può assumere il valore nullo. Mentre l'Equazione 3.2 è un vincolo ricavato dalla Formula 3.1

   $$i < 0 = false \wedge i = 0 \Rightarrow (-1 < i) = true$$

   Formula 3.1

   1.3. Nega l'ultimo vincolo :

   $$\neg(-1 < i) \Longrightarrow (i < -1) = true \Rightarrow i = Int.Min$$

   Formula 3.2

   dalla Formula 3.2 il solver ha risolto in $i = Int.Min$, il più basso numero intero;

[22] Output : In questo caso si intende il canale utilizzato dai programmi per visualizzare sullo schermo messaggi per l'utente. Normalmente chiamato Standard Output si differenzia dallo Standard Input che al contrario è quello utilizzato per ricevere i comandi dall'utente inseriti attraverso dispositivi esterni come la tastiera.

2. Esecuzione del secondo test ($Test$ 2), con

   i = Int.Min

   Bar(i=Int.Min,j=0)

   Nell'esecuzione di $Test$ 2 Pex:

   2.1. Esplora la riga 3 e si ferma alla riga 5;

   2.2. Ricava i seguenti 3 vincoli:

   $$return\ target\ != \ (ClassMethod)null;$$

   Equazione 3.3

   $$return\ target\ != \ (ClassMethod)null\ \&\&\ i\ <\ 0;$$

   Equazione 3.4

   $$\text{return target} != \ \text{(ClassMethod)null} \ \&\& \ \text{i} < 0 \ \&\& \ \text{j} != \ 123;$$

   Equazione 3.5

   L'Equazione 3.5 ricavata dal nodo decisionale in riga 2 come evidenziato nella Formula 3.3

   $$(\text{j} == 123) = false\ \wedge\ j\ =\ 0\ \Rightarrow\ (j\ !=\ 123)\ =\ true$$

   Formula 3.3

   2.3. Nega l'ultimo vincolo:

   $$\neg(j\ !=\ 123)\ \Rightarrow\ (j\ ==\ 123)\ =\ true$$

   Formula 3.4

   dalla Formula 3.4 il solver ha risolto in *j = 123*

3. Esecuzione del terzo test ($Test$ 3), con j = 123

   Bar(i=Int.Min,j=123)

   Nell'esecuzione di $Test$ 3 Pex:

   3.1. Esplora le righe 3,5 e 6 scrivendo nello standard output "$Linea$ 6";

   3.2. Non trova più nuovi vincoli da negare e si ferma.

Questo breve esempio ha voluto dare al lettore una semplice sequenza di operazioni per poi approfondire nei capitoli successivi il comportamento di Pex in modo dettagliato.

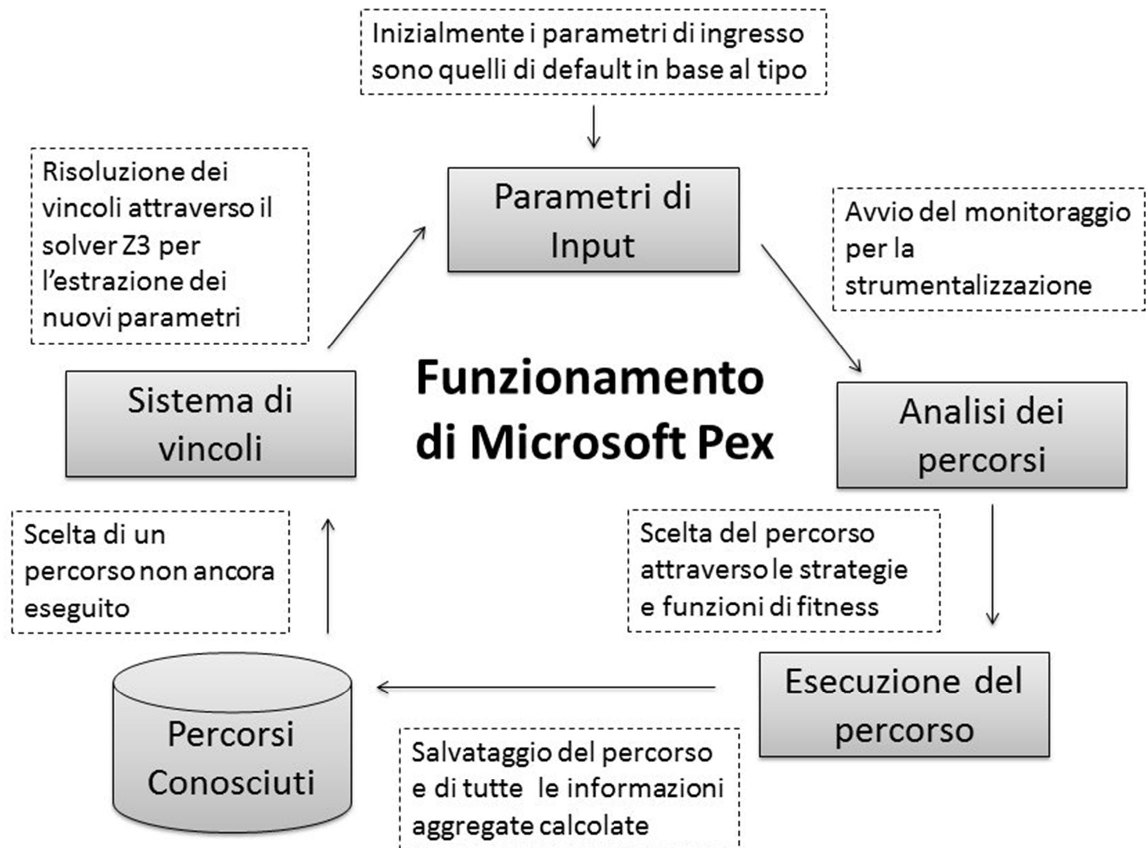


Figura 3.1 - Funzionamento di Pex

3.2.2.2. Esecuzione simbolica dinamica

Pex genera gli input dei test da eseguire utilizzando la tecnica di “esecuzione simbolica”. L'esecuzione simbolica funziona in modo simile all'esecuzione del programma. Quando una variabile del programma è aggiornata ad un nuovo valore durante l'esecuzione, questo nuovo valore diventa un valore simbolico da utilizzare. Quando il programma esegue un ramo decisionale in cui la condizione è un'espressione della variabile simbolica, l'esecuzione simbolica deve considerare tutte le possibili decisioni condizionate da tutti i possibili valori che la variabile può assumere.

Per ciascun percorso esplorato, sulla base dei nodi decisionali analizzati, sono raccolti tutti i vincoli che hanno portato ad eseguire il percorso corrente. Un risolutore di vincoli viene utilizzato per decidere la fattibilità dei percorsi individuati e per ottenere gli ingressi di

prova che hanno permesso i singoli percorsi. I vincoli raccolti determineranno anche il successivo ramo da visitare che dovrà essere diverso dal percorso precedente. In questo modo alla fine dei test generati tutti i percorsi possibili saranno stati eseguiti.

L'esecuzione simbolica dinamica estende la convenzionale esecuzione simbolica statica con informazioni aggiuntive che vengono raccolte in fase di esecuzione, le quali rendono l'analisi più precisa.

Di seguito l' algoritmo di esecuzione simbolica dinamica attuato in Pex.

$J := 0$, J *è l'insieme dei parametri di Input già analizzati*

Inizia ciclo:

Calcola parametri di input $i \notin J$,
se non hai il risultato vai a Fine ciclo

Output i,

Esecuzione $P(i)$, *salvataggio della condizione del percorso* $C(i)$

$J := J \cup C$, C *è composta da* $\{i \text{ oppure } C(i)\}$

Fine ciclo

Algoritmo 3.2 - Esecuzione simbolica dinamica adottata da Pex

La scelta degli ingressi ad ogni iterazione del ciclo, definisce l'ordine di esecuzione dei diversi percorsi possibili del programma.

L'obiettivo finale di Pex è quello di scoprire tutte le istruzioni raggiungibili nel modo più veloce possibile. Altri strumenti che applicano questa tecnica effettuano una ricerca esaustiva di tutte le esecuzioni percorribili in un ordine predeterminato, entro limiti definiti dalle dimensioni e strutture del dato in ingresso forniti dall'utente. Pex permette all'utente di definire la strategia di esecuzione ed alcuni tempi massimi di esecuzione, come per esempio il tempo massimo entro cui il solver deve risolve l'insieme di vincoli passati da Pex.

3.2.2.2.1. Instrumentation

Non è raro che i programmi sviluppati in .NET utilizzano puntatori aritmetici per accedere direttamente alla memoria per motivi di prestazioni o nella maggior parte delle volte per interagire con altri programmi non sviluppati in .NET come i driver, o accessi alle API[23] di Windows per motivi di ereditarietà.

Mentre gli algoritmi di analisi statica di esecuzione simbolica non possono utilizzare alcuna informazione su l'ambiente in cui il programma è avviato, l'esecuzione simbolica dinamica è in grado di riutilizzare le informazioni dinamiche che osserva durante l'esecuzione. Vengono osservate locazioni di memoria, accessibili attraverso i puntatori, e per esempio i dati passati tra il programma analizzato e l'ambiente di esecuzione.

Pex può "monitorare" codice realizzato in un linguaggio .NET tramite la "strumentazione" (tecnicamente chiamata *instrumentation*) del codice. Pex si inserisce nel livello applicativo chiamato ".NET profiling API", prima che il linguaggio sia compilato dal Microsoft Intermediate Language (MSIL) in codice binario, attraverso Just-In-Time Compiler, Pex riscrive le istruzioni, applicando le seguenti ulteriori operazioni:

- Costruisce rappresentazioni simboliche delle operazioni eseguite sulle variabili sostituendo quelle del programma;
- Mantiene e sviluppa una rappresentazione simbolica dello stato del programma intero in qualsiasi momento;
- Tiene traccia delle condizioni visitate.

Pex non conosce quale sia il linguaggio utilizzato ad alto livello, ma al momento della scrittura di questo documento solo il C# è supportato.

Inizialmente Pex considera tutto il codice non strumentalizzato poiché il processo del monitoraggio è molto dispendioso in termini di risorse. Durante il processo di analisi Pex quando trova il richiamo ad un metodo esterno al namespace che si sta analizzando, lo considera come processo esterno e generando un warning con il messaggio "*Uninstrumented method*", se lo si vorrà analizzare sarà necessario dichiararlo come

[23] API : Le "Application Programming Interface" (letteralmente "Interfaccia di Programmazione di un'Applicazione") indicano la struttura del codice, namespace, classi e metodi ed indicazioni su come utilizzarli.

"*Instrumented Method*". Nei prossimi capitoli verrà spiegato come effettuare questa operazione.

Concludendo, il processo di monitoraggio permette di intervenire sulle chiamate verso gli ambienti esterni in modo da poter non effettuare la reale chiamata ma poter completare il singolo test secondo le esigenze. Il come effettuare l'isolamento verrà spiegato nei successivi paragrafi.

3.2.2.3. Strategie

Si definisce strategia il modo in cui partendo da un nodo da visitare, si determina quale sarà il successivo. La strategia viene applicata nel ciclo definito dall'Algoritmo 3.2, determinando l'ordine di navigazione dei singoli nodi.

Pex, seguendo la teoria dei grafi e gli algoritmi di navigazione degli stessi, può utilizzare le seguenti strategie di base:

- *Breadth-first o navigazione in ampiezza*: favorisce la visita dei nodi iniziali per poi scendere in profondità;
- *Depth-first o navigazione in profondità*: partendo dai nodi iniziali, scende nei nodi in profondità figli, risale e ridiscende fino al completamento. Questa strategia visita prima i nodi in profondità;
- *Iterative deepening o ricerca in profondità controllata*: effettua una ricerca in profondità risalendo non alla fine del ramo ma ad un limite di profondità definito. La ricerca parte dal livello zero (i nodi iniziali) e prosegue aumentando in modo unitario il livello della profondità da analizzare fino ad arrivare al massimo livello. Esso può non essere raggiunto se l'obiettivo proposto alla ricerca viene raggiunto prima;
- *Random o navigazione casuale*: l'ordine non è definito ed i nodi vengono visitati casualmente senza ripetizione.

Non specificando quale strategia utilizzare nel singolo test o suite di test, Pex dalla versione 0.7.30916.0 utilizza un algoritmo che sceglie la strategia da utilizzare ad ogni iterazione, basandosi su quelle di base. Questa nuova strategia è chiamata *Fitnex* [25].

3.2.2.3.1. Strategia Fitnex

La strategia Fitnex è nata dall'esigenza di adottare una strategia differente per ogni iterazione, basandosi sull'obiettivo di generare parametri di input per i test successivi. Si è capito che non tutti i nodi potrebbero avere una uguale importanza. I parametri di input vengono ricavati dai nodi decisionali, si darà quindi più importanza a quelli che avranno una dipendenza diretta con tali parametri.

Per esempio considerando una funzione con la seguente firma :

$$public\ bool\ f\ (int\ x, int\ []\ y)$$

Esempio 3.2

e considerando due nodi decisionali presenti nella funzione:

$$if\ (x\ ==\ 110)$$

Esempio 3.3

$$if\ (y[i]\ ==\ 15)$$

Esempio 3.4

la relazione Esempio 3.4 è indiretta poiché condizionata anche dalla variabile i oltre che da y.

La scelta di navigazione al nodo successivo non si basa solo sulla risoluzione dei vincoli, ma anche sulle informazioni strutturali del programma e sui percorsi precedentemente eseguiti.

Questa nuova strategia, definisce una ricerca dei nodi da visitare guidata alleviando i problemi della esecuzione simbolica dinamica composta da una ricerca esaustiva o casuale.

La strategia utilizza una funzione di idoneità definita come "*funzione di fitness*"[24] che misura "quanto è vicino" un percorso esplorato all'obiettivo di copertura massima del test. È anche definita una funzione di "*guadagno di fitness*", utilizzata per ogni nodo del ramo esplorato per poter assegnare priorità di navigazioni diverse per ogni singolo nodo e ramo.

[24] Funzione di fitness : Letteralmente “buona forma” o idoneità in quanto effettua una misura.

Di seguito l'algoritmo di questa strategia.

1. Viene calcolato un "*valore di fitness*" per ogni percorso testato P_i

$$VF_i = f(P_i)$$

Formula 3.5

2. Viene calcolato un "*guadagno di fitness*" per ogni ramo *b* facente parte del percorso P_i

$$Gb = fGain(b)$$

Formula 3.6

3. Durante l'esplorazione del percorso P_i si darà alta priorità di esplorazione ad un nodo di ramificazione *b* facente parte di P_i con un miglior (più alto) guadagno di fitness *Gb* rispetto ad un percorso *P* con un miglior (più basso) valore di fitness *VF* .

Per ogni esplorazione:

1. Per ogni nodo *bn* del ramo *b* nel percorso P_i con valore di fitness FV_i , viene definito un nuovo percorso negando (azione chiamata *flipped*, letteralmente "agitato") il singolo nodo e definendo un nuovo percorso P_{i+1} con un valore di fitness VF_{i+1} . Di seguito l'insieme di regole utilizzate per calcolare FV_i in funzione del predicato e del valore del nodo analizzato

N. Regola	Predicato	Valore se Vero	Valore se falso
1.	f(a == b)	0	|a – b|
2.	f(a > b)	0	(b – a) + K
3.	f(a >= b)	0	(b – a)
4.	f(a < b)	0	(a – b) + K
5.	f(a <= b)	0	(a – b)

Tabella 3.1 – Valori della funzione di Fitness FV_i assegnato ai nodi

2. Per ogni nodo *bn* negato, viene calcolato il guadagno

$$Gbn = fGain(bn) = (FV_i - FV_{i+1})$$

Formula 3.7

Se il nodo viene visitato per la prima volta, il guadagno è massimo

$$Gbn = fGain(bn) = FV_i$$

Formula 3.8

Se Gbn è negativo allora, non avendo un guadagno dalla negazione di bn, non è desiderabile effettuarla.

3. Per ogni ramo *b* viene calcolato il guadagno medio dei *bn* di cui è composto

$$Gb = fGain(b) = Avg(fGain(bn))$$

Formula 3.9

Per dare priorità ad ogni ramo *b* facente parte del percorso *P* si associa un valore di fitness definito come

$$VFb = (VF - Gb)$$

Formula 3.10

Algoritmo 3.3 – Procedura adottata dalla strategia di Fitness

Come descritto nel punto 1. dell'esplorazione, il valore derivato dalla funzione di fitness viene calcolato sulla base dello stato dei predicati dei nodi del percorso visitato. Nella Tabella 3.1 viene illustrato, come dato un predicato di destinazione, venga associato un valore di fitness FV_i, derivato dal suo attuale stato (vero se già visitato, falso se è la prima visita), dai sui termini di cui è composto (*a* e *b*) e da un valore costante *K* .

Basandosi sul concetto che Pex non necessita di predicati compositi con operatori logici come $\wedge$ ed $\vee$ poiché opera su istruzioni semplici di .NET ed a questo livello eventuali relazioni composte sono già state scomposte, la funzione di fitness non le supporta.

Di seguito la spiegazione di come l'Algoritmo 3.3 – Procedura adottata dalla strategia di Fitness, viene applicato all'Esempio 3.1 facendo riferimento al paragrafo 3.2.2.10 - Esempio di esplorazione, nel quale veniva evidenziato come calcolare i valori di input dei test generati.

Si definiscono i seguenti nodi condizionati:

$C_1: (i < 0)$

$C_2 : (j \ == \ 123)$

vengono eseguiti i seguenti test :

1. $Test$ 1:

   $Bar(i = 0, j = 0)$

   $P_1 \ : \ C_1(falsa)$

   1.1. Considerando C_1 viene applicata la regola 4 dalla Tabella 3.1, definendo $K \ = \ 1$

   $$FV_1(C_1) \ = \ (a \ - \ b) \ + \ K \ = \ (0 \ - \ 0) + 1 \ = \ 1$$

   1.2. Si applica la Formula 3.8

   $$G_1C_1 \ = \ fGain(bn) = FV_1(C_1) = \ 1$$

   1.3. Si applica la Formula 3.10

   $$VFC_1 \ = \ (VF \ - \ Gb) = \ (\ VFC_1 - \ G_1C_1) = \ (0 \ - 1) = \ -1$$

   1.4. Si nega la condizione C_1 , passando il vincolo al risolutore il quale ricaverà un parametro per il test successivo;

2. $Test$ 2:

   $Bar(i = Int.Min, j = 0)$

   $P_2 \ : \ C_1 \ (vera) \ \ C_2(falsa)$

   2.1. Considerando C_1 viene applicata la regola 4 dalla Tabella 3.1

   $$FV_2(C_1) \ = \ 0$$

   2.2. Si applica la Formula 3.7

   $$G_2C_1 \ = \ fGain(bn) = \left(FV_1(C_1) \ - \ FV_2(C_1)\right) = \ (1 \ - \ 0) = \ 1$$

   2.3. Si applica la Formula 3.10

   $$VFC_1 \ = \ (VF \ - \ Gb) \ = \ (\ VFC_1 \ - \ G_2C_1) \ = \ (1 \ - 1) \ = \ 0$$

2.4. Considerando C_2 viene applicata la regola 1 dalla Tabella 3.1

$$FV_1(C_2) = |a - b| = |0 - 123| = 123$$

2.5. Si applica la Formula 3.8

$$G_1C_2 = fGain(bn) = FV_1(C_2) = 123$$

2.6. Si applica la Formula 3.10

$$VFC_2 = (VF - Gb) = (VFC_2 - G_1C_2) = (0 - 123) = -123$$

2.7. Considerando VFC_1 e VFC_2 viene negata la condizione C_2, come spiegato nel punto 3 dell'Algoritmo 3.3 – Procedura adottata dalla strategia di Fitness nell'esplorazione

$$Min(VFC_1, VFC_2) \Rightarrow \neg C_2$$

2.8. Passando il vincolo al risolutore, egli ricava un parametro per il test successivo;

3. $Test$ 3:

$Bar(i = Int.Min, j = 123)$

$P_2 \, : \, C_1 \, (vera) \;\; C_2(vera)$

3.1. Considerando C_1 viene applicata la regola 4 dalla Tabella 3.1

$$FV_3(C_1) = 0$$

3.2. Si applica la Formula 3.7

$$G_3C_1 = fGain(bn) = \big(FV_2(C_1) - FV_3(C_1)\big) = (0 - 0) = 0$$

3.3. Si applica la Formula 3.10

$$VFC_1 = (VF - Gb) = (VFC_1 - G_3C_1) = (0 - 0) = 0$$

3.4. Considerando C_2 viene applicata la regola 1 dalla Tabella 3.1

$$FV_2(C_2) = 0$$

3.5. Si applica la Formula 3.7

$$G_2C_2 = fGain(bn) = \left(FV_1(C_2) - FV_2(C_2)\right) = (123 - 0) = 123$$

3.6. Si applica la Formula 3.10

$$VFC_2 = (VF - Gb) = (VFC_2 - G_2C_2) = (-123 - 123)) = -256$$

3.7. Poiché sono state negate tutte le condizioni almeno una volta, l'esplorazione si ferma.

Sono quindi stati effettuati i tre percorsi $P_1,\ P_2\ e\ P_3$ in base ai valori di fitness trovati.

La Formula 3.9 non è mai stata applicata in quanto nessun ramo era formato da più di un nodo.

3.2.2.4. Risolutore dei vincoli

Ogni percorso è rappresentato da un sistema di vincoli, come evidenziato nel precedente paragrafo 3.2.2.1 - Esempio di esplorazione. Pex, per calcolare i parametri per la generazione del test successivo da eseguire per l' esplorazione, utilizza un risolutore (anche chiamato "*solver*").

Un risolutore, datogli un problema come insieme di equazioni (o vincoli) composte da incognite, restituisce l'insieme di valori associati alle incognite che, se esiste soluzione, risolvono il problema. Durante l'esplorazione i vincoli cambiano continuamente attraverso l'azione di *flipped*, generando sempre nuovi valori.

Il solver adottato da Pex è Microsoft Z3 [24], prodotto dal gruppo di ricerca Microsoft ed attualmente utilizzato anche in molti altri progetti.

Z3 è un solver SMT[25] ("Satisfiability Modulo Theories", letteralmente "soddisfacibilità teorica di un modulo") [26]. Al suo interno contiene un risolutore di vincoli lineare, pertanto i vincoli non lineari non vengono risolti.

Pex esplorando il metodo seguente:

```
1  public int MultiplyInt(int a, int j)
2  {
3     if (a * a == 2)
4     {
5        if (j == 123)
6        {
7           a = 2;
8        }
9     }
10    else
11       a = 3;
12
13    return 3;
14 }
```

Esempio 3.5 – Metodo con una condizione non lineare di tipo numerico intero (*int*)

genera un solo test

$MultiplyInt(0,0)$

utilizzando i valori di default ed ignorando la possibilità di negare la condizione

$if(a * a) == 2$

Anche i numeri a virgola mobile non sono supportati (i tipi *float*, *double* e *decimal*)[26], come dimostrato nel successivo esempio.

[25] Solver SMT : Un risolutore "Satisfiability Modulo Theories", letteralmente "soddisfacibilità teorica di un modulo" si differenzia da un comune SAT solver, "Satisfiability Arithmetic Theories solver" letteramente "risolutore di soddisfacibilità teorica aritmetica" dal fatto che SAT risolve solo sistemi di equazioni binarie o booleane, mentre SMT supporta non solo aritmetica ad una dimensione fissa, ma anche per moduli di vettori, matrici, quantificatori ed altre funzioni modulari teoriche di primo ordine.

[26] Tipi float, double e decimal : per tipi si intende la dimensione e tipologia di dato associato alle variabili. In questo caso specifico sono tipi numerici con decimali.

```
1  public float MultiplyFloat(float a, float j)
2  {
3    if (a * a == 2)
4    {
5      if (j == 123)
6      {
7        a = 2;
8      }
9    }
10   else
11     a = 3;
12
13   return 3;
14 }
```

Esempio 3.6 – Metodo con condizione non lineare di tipo “float”

Pex durante l’esplorazione genera dei *warning* (messaggi di avviso) scrivendoli del file di log[27]:

1. $00{:}00{:}03.0 > MultiplyFloat(ClassMethod, Single, Single)$
2. $[instrumentation]\ testability\ issue\ in\ floating\ point\ equality$
3. $[instrumentation]\ limitation\ in\ floating\ point\ multiplication$
4. $[instrumentation]\ limitation\ in\ floating\ point\ conversion$

Da cui si evince che nel test c’è stato un problema di uguaglianza in virgola mobile (riga 2.) e delle limitazioni, sempre in virgola mobile, per una moltiplicazione (riga 3.) ed in fine una conversione (riga 4.) .

Per risolvere il problema della risoluzione dei vincoli con l'utilizzo di numeri con la virgola, sono nati nuovi progetti come FloPSy [27] estensione di Pex .

Dopo aver installato l'estensione, aggiungendo l'attributo

$[PexCustomArithmeticSolver]$

nei metodi o classe da analizzare, Pex raccoglie tutte le variabili (di tipo float, double o decimal) che appaiono nelle condizione del percorso, che non potrebbero essere gestite con il risolutore Z3, passandoli alla classe di estensione. Essa, $PexCustomArithmeticSolver$, utilizza un algoritmo di ricerca euristica per cercare di trovare i valori per queste variabili

[27] File di log : Letteralmente tronco di legno, nell’informatica i file di log sono file di testo dove vengono salvati messaggi descrittivi sulla cronologia degli eventi dell’applicazione.

di input che soddisfano i vincoli attraverso alcuni solver interni a FloPSy, fornendo a Pex i soli valori da utilizzare.

3.2.3. L’isolamento del codice

Una unit test per definizione effettua il test con il codice in isolamento. Questo significa che tutte le dipendenze (basi di dati, collegamenti di rete ed accessi a dispositivi esterni di archiviazione o file) devono essere “nascoste” al test, per esempio attraverso un livello di astrazione. Durante il test il livello di astrazione deve sostituire l'ambiente esterno facendo credere che esso funziona correttamente poiché non è in questo contesto che si vogliono verificare i comportamenti dell'ambiente esterno ma ci si vuole focalizzare solo sul codice testato.

Nel mondo della programmazione, è un problema comune testare le applicazioni che utilizzano database, senza fare l’accesso al database reale. Va anche considerato che un aspetto essenziale della unit test è quello di testare una caratteristica alla volta, sapendo esattamente ciò che si sta provando e dove potrebbero esserci eventuali problemi. Una soluzione è l'utilizzo di un oggetto fittizio per "falsificare" la comunicazione con il mondo esterno.

3.2.3.1. Tipi di oggetti fittizi

L'oggetto fittizio, nel tempo si è evoluto cambiando anche denominazione ed arrivando alle seguenti definizioni di seguito riportate in ordine cronologico :

1. *Stub (letteralmente estirpare)*: rappresenta la porzione di codice utilizzata nel test attraverso l'implementazione dei metodi di una interfaccia richiamati nell'esecuzione. In produzione ci sarà l'implementazione reale che comunica con l'ambiente esterno;
2. *Fake (letteralmente falsificare)*: evoluzione dello *Stub*, ritorna dei valori predefiniti per ciascun metodo dell'interfaccia implementata, i valori sono dinamici in base all'esecuzione del programma. Può anche essere chiamato *Dummy* (letteralmente fittizio) se rimpiazza completamente l'implementazione utilizzata in produzione nella sola fase di test;

3. *Mock (letteralmente beffa)*: evoluzione del *Fake*, permette il controllo diretto dei metodi, essi potrebbero essere invocati sulle risorse esterne ma "pilotati" dall'oggetto Mock. L'oggetto viene creato solo in fase di testing ma a differenza dell'oggetto di tipo *Dummy* l'oggetto può contenere maggiori funzionalità rispetto a quello di produzione per poter facilitare il testing.

Esistono molti framework per la creazione di oggetti *Mock* per qualsiasi linguaggio di programmazione.

3.2.3.2. Microsoft Moles

Microsoft Moles è il framework sviluppato da Microsoft per fronteggiare il tema dell'isolamento del codice in fase di testing. Pex ha adottato questo framework per poter aiutare lo sviluppatore di unit test a creare il livello astratto per rendere indipendente la unit test dall'ambiente esterno.

Moles permette la sostituzione di qualsiasi metodo chiamato in fase di test con una implementazione sostituita da un metodo delegato[28] che restituirà il risultato adatto per il test.

Il framework permette di creare :

- *Oggetti di tipo Stub*: comporta la riscrittura del codice attraverso l'implementazione di una qualsiasi interfaccia .NET;
- *Oggetti di tipo Mock*: attraverso l'*instrumentation*, come discusso nel paragrafo 3.2.2.2.1 - Instrumentation, viene utilizzata come alternativa al refactoring[29] quando esso non è possibile. Usando il processo di monitoraggio, gli oggetti strumentalizzati, possono essere "deviati" dal loro comportamento "normale".

Concettualmente un oggetto Stub o Mock si potrebbe creare semplicemente facendo una nuova implementazione del metodo oppure dell'oggetto da sostituire richiamando il nuovo

[28] Metodo delegato: Un metodo delegato è strettamente una implementazione C#, molto simile a un puntatore a una funzione in C e C++, esso è orientato agli oggetti, indipendente dai tipi, con solo l'accesso alla memoria allocata per l'esecuzione in corso.

[29] Refactoring : Letteralmente ricostruzione, si intende il processo di miglioramento del codice riscrivendolo e riprogettandolo internamente senza modificarne il comportamento esterno e quindi, per esempio, senza variare la firma di un metodo.

in fase di test ma sarebbe necessario predisporre il codice con il modello "Dependecing Injection" (letteralmente "dipendenza iniettata") predisponendo:

- La componente dipendente (l'oggetto che può variare);
- La dichiarazione delle dipendenze del componente, definite come interfaccia;
- Un *injector* (chiamato anche provider o container, rispettivamente iniettore, fornitore e contenitore) che crea, a richiesta, le istanze delle classi che implementano la realtà da utilizzare.

Definito l'oggetto da sostituire, Moles ricompila la libreria dichiarando dei nuovi oggetti e metodi che si potranno richiamare e sostituire con la tecnica dei metodi delegati. Per convenzione ciascun oggetto di tipo Stub sarà:

1. Creato in un nuovo namespace derivato dal vecchio con l'aggiunta della parola Moles
   $NamespaceVecchio.Moles$
2. Anteposta la lettera "S" al nome dell'interfaccia.

Per esempio, l'interfaccia $MyNamespace.MyInterface$ verrà denominata $MyNamespace.Moles.SMyInterface$.

Un oggetto di tipo Mock invece sarà:

1. Creato in un nuovo namespace derivato dal vecchio
   $NamespaceVecchio.Moles$
2. Anteposta la lettera "M" al nome della classe;
3. Nel test sarà necessario definire un attributo per definire l'utilizzo della strumentalizzazione. Per esempio se si utilizza il framework MSTest si dovrà aggiungere l'attributo $HostType$

```
[TestMethod]
[HostType("Moles")]
public void UnitTest()
{
}
```

esso dipende dal framework utilizzato.

Per esempio, il metodo $MyNamespace.MyClass.Test()$ diventerà $MyNamespace.Moles.MMyClass.Test$.

Per utilizzare il framework Moles è necessario seguire i seguenti passi:

1. In Visual Studio, selezionare la libreria per cui si vuole utilizzare l'oggetto analogo "fittizio" tra quelle referenziate e dopo aver premuto il tasto destro del mouse sulla selezione, dal menù selezionare $Add\ Moles\ Assembly$. Verrà aggiunto un file XML di progetto
$nomelibreria.moles$
esso è un file di configurazione dove è possibile specificare alcune personalizzazioni per filtrare solo alcuni oggetti e non tuta la libreria;
2. Ricompilare il progetto.
L'applicazione "*MSBuild Moles*" compilerà in base alle configurazioni del file $nomelibreria.moles$ aggiungendo i nuovi oggetti ed anche i riferimenti agli assembly necessari in modo automatico, come ad esempio la libreria di Moles $Microsoft.Moles.Framework.dll$
3. Nel test utilizzare gli oggetti di Moles quando necessario.

3.2.3.3. Pex e Moles

Durante la generazione dei test attraverso l'integrazione in Visual Studio, Pex suggerisce quando utilizzare Moles poiché riscontra una dipendenza verso l'esterno.

Nella finestra dei risultati, nella sezione "$Testabiltiy$", Pex accoda tutti i metodi per cui suggerisce di usare Moles attraverso il suggerimento "$Mole\ Type$".

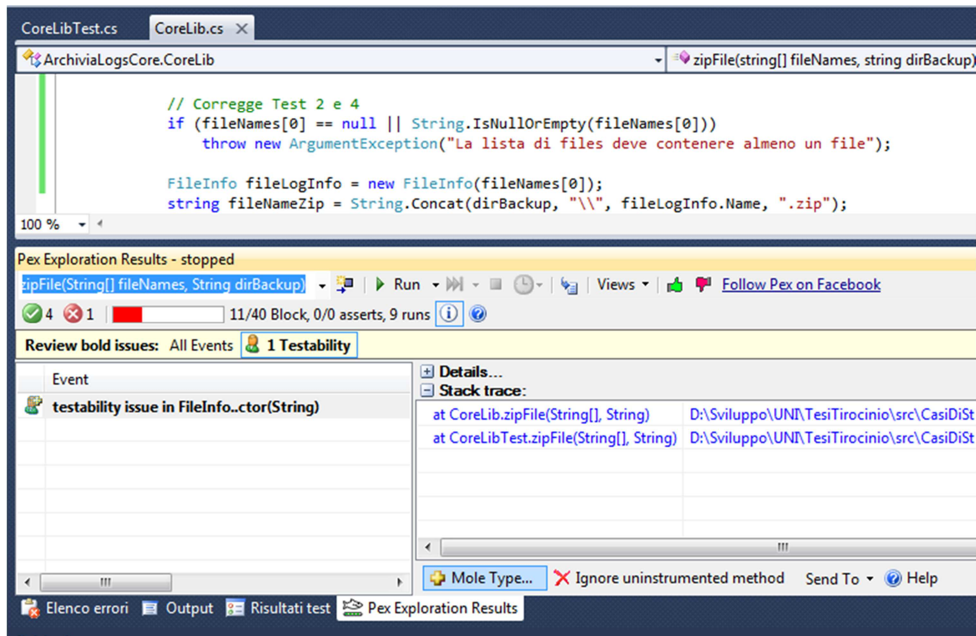


Figura 3.2- Applicazione del suggerimento "Mole Type"

Applicando il suggerimento Pex aggiunge nel PUT la chiamata ad un metodo “*Prepare*" (letteralmente preparare) predisposto per l'utente per “preparare” gli oggetti fittizi creati attraverso Moles.

Considerando il seguente frammento di codice come esempio su cui Pex si è fermato nell’esplorazione:

$$FileInfo\ fileLogInfo\ = new\ FileInfo(fileNames[0]);$$

Esempio 3.7 - Oggetto esterno

Pex effettuerà i seguenti passaggi :

1. Aggiunge il file $mscorlib.moles$

$$< Moles\ xmlns = "http://schemas.microsoft.com/moles/2010/" >$$

$$< Assembly\ Name = "mscorlib"\ ReflectionOnly = "true"\ />$$

$$</Moles>$$

2. Aggiunge la classe di preparazione $System.IO.Preparations.FileInfoPreparation$

```
// <copyright file = "FileInfoPreparation.cs" company = "Microsoft"
        > Copyright © Microsoft 2012 </copyright >

using System;

using Microsoft.Pex.Framework;

using System.IO.Moles;

using System.IO;


namespace System.IO.Preparations

{

  /// < summary > Contains a method to prepare the type FileInfo
        </summary >

  public static partial class FileInfoPreparation

  {

   /// < summary >
> Prepares the environment (and the moles)before executing any method

of the prepared type FileInfo </summary >

     [PexPreparationMethod(typeof(FileInfo))]

     public static void Prepare()

     {

       MFileInfo.BehaveAsCurrent();

       // TODO: use Moles to replace API that call into the environment

     }

  }

}
```

Esempio 3.8 - Metodo Prepare da implementare

3. Aggiunge il richiamo del metodo nella PUT

```
public void eseguiSpostamento18()
{
...
  FileInfoPreparation.Prepare();
...
```

```
}
```

Riavviando l'esplorazione Pex si aspetterà che il metodo dove si è applicato il suggerimento di utilizzare Moles sia stato sostituito con una implementazione alternativa. Considerando l'Esempio 3.7 - Oggetto esterno, si potrebbero implementare nel metodo "*Prepare*" le seguenti righe di codice :

```
string fileName = "AdapterWS_Errors.2012-01-01";

string dirFileName
        = @"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
        \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
        \Tests";


MFileInfo.BehaveAsCurrent();

MFileInfo.ConstructorString = (@this, a) =>

{

};

MFileInfo.AllInstances.NameGet = (@this) =>

{

return fileName;

};

MFileInfo.AllInstances.DirectoryNameGet = (@this) =>

{

return dirFileName;

};
```

Esempio 3.9 - Esempio di codice isolato con Moles

Pex potrà quindi proseguire nella generazione dei test, diversamente verrà generata una eccezione di tipo *Moles.Framework.Moles.MoleNotImplementedException*.

L'alternativa al primo suggerimento potrebbe essere utilizzare il secondo suggerimento "*Ignore testability issue*" (letteralmente "ignora il problema di testabilità") ma in questo modo non si potrà raggiungere la copertura al del codice al 100%.

3.2.4. Avvio di Pex ed utilizzo

Pex può essere utilizzato sia attraverso il componente aggiuntivo installato in Visual Studio oppure da linea di comando.

3.2.4.1. Avvio di Pex da dentro l'ambiente di sviluppo Microsoft Visual Studio

Dopo aver installato Pex, in qualsiasi progetto di Visual Studio, cliccando con il tasto destro del mouse nel codice sorgente di una classe, ci sarà una nuova voce di menu "*Run Pex*" da utilizzare per l'avvio dell'esplorazione. È anche possibile avviarla sull'intero progetto facendo il tasto destro sul nome del progetto nel riquadro "*Esplora Soluzione*".

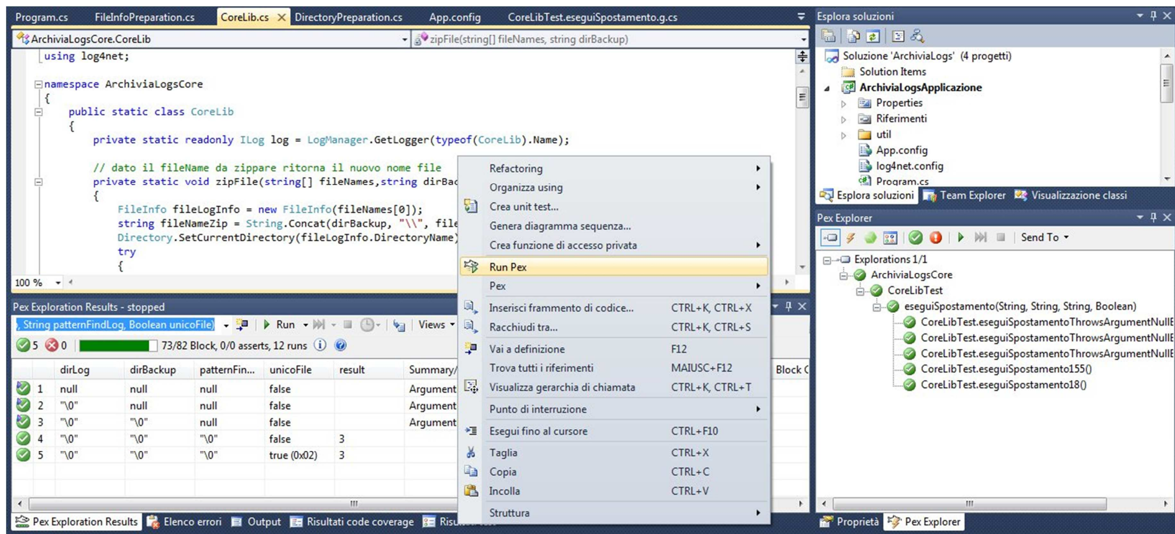


Figura 3.3 – Avvio di Pex all'interno di Microsoft Visual Studio

Pex mette a disposizione diverse finestre di controllo.

La finestra "*Pex Explorer*" (letteralmente "esploratore Pex"), normalmente sulla destra, visualizza lo stato dell'ultima esplorazione eseguita o in esecuzione, permettendo la navigazione dei test generati. È abbinata alla finestra "*Pex Exploration results*" (letteralmente "risultati dell'esplorazione Pex"), normalmente in basso, la quale visualizza tutti i dettagli per ogni singolo test generato.

"*Pex Exploration results*" è la finestra fondamentale per lo sviluppatore. Essa è composta da una prima barra contenente: dei pulsanti per il controllo dell'esecuzione, altri

per avere il dettaglio sul metodo testato, ed altri ancora per l'importazione ed esportazione dei risultati, infine un menu a tendina con diversi comandi avanzati. Una successiva barra viene creata in fase di esplorazione con il sommario dei risultati dei test generati, i totali dei test passati, i totali dei test falliti, il numero di blocchi analizzati, il numero dei blocchi totali ed il numero di esecuzioni eseguite. Una terza barra viene creata ad ogni esplorazione per segnalare la presenza di metodi richiamati che non sono testabili, strumentalizzati oppure dichiarati come esterni. Cliccando sulla segnalazione si può guardare il dettaglio dell'evento che l’ha scatenato, i dettagli della eventuale eccezione di errore e la parte di codice sorgente interessata. In basso a destra sotto i dettagli, vi saranno dei suggerimenti per poter proseguire nello sviluppo della suite di test automatici.

Infine in una tabella vengono visualizzati i dettagli del metodo analizzato, con un insieme di righe rappresentanti il dettaglio della singola esecuzione. La prima icona ha molti significati tra cui se il test è passato o fallito, se il test è già stato salvato oppure no. Nel resto delle colonne è possibile vedere i valori di input utilizzati, il risultato ottenuto, messaggi di errori, percentuale di copertura ed *Assertion* (letteralmente affermazione) impostate nel test e passate.

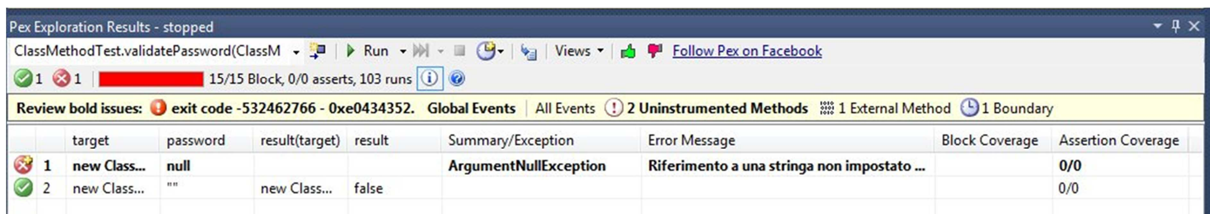


Figura 3.4 – Finestra Pex Exploration Results

Dopo aver eseguito la prima volta l'esplorazione, Pex suggerisce di salvare i test generati. Il wizard[30] da seguire, crea un nuovo progetto di test da poter utilizzare e salvare le PUT da poter riutilizzare nei test di regressione. In questo modo si salvano anche i suggerimenti dati ed applicati durante l'esplorazione .

[30] Wizard : Letteralmente mago, nell’informatica si intende quel processo guidato formato da più passi rappresentati da finestre differenti al fine di ottenere un risultato finale gradualmente.

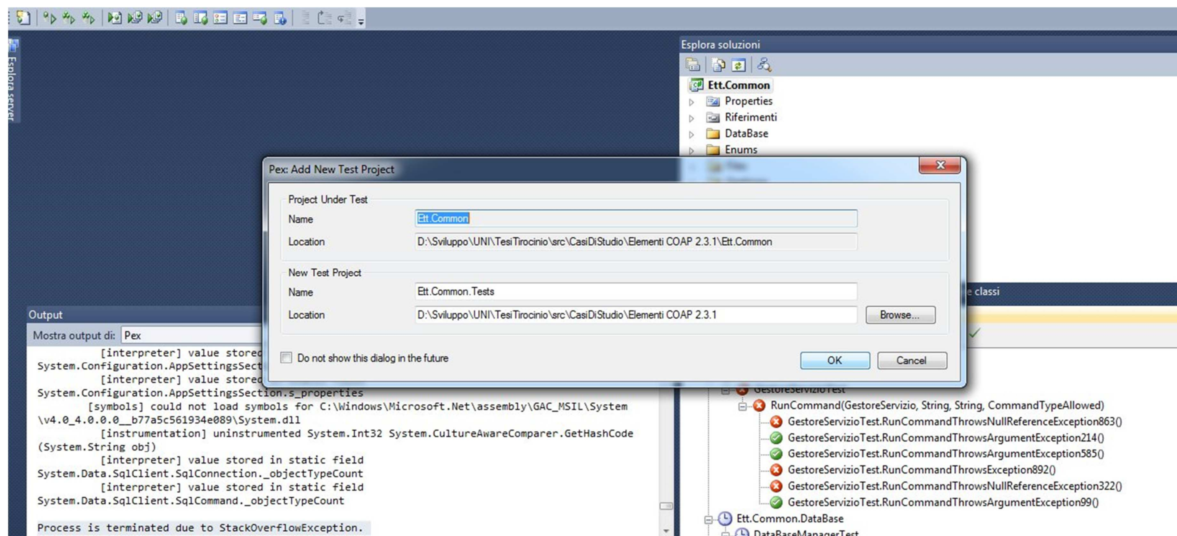


Figura 3.5 - Procedura guidata per la creazione del progetto di test

3.2.4.2. Avvio da linea di comando

L'esplorazione da linea di comando viene spesso avviata quando si vuole utilizzare particolari configurazioni dell'esplorazione stessa, oppure quando non si hanno a disposizione il codice sorgente ma solo l'eseguibile o la libreria dll, oppure quando si vuole salvare la reportistica generata sempre nella stessa cartella, oppure quando si pianificano dei test automatici insieme ad un sistema esterno che richiama solo l'avvio in modo automatizzato.

Di seguito il modo più semplice per avviare l'esplorazione dopo aver aperto una finestra di comandi MS-DOS[31] :

$$pex.exe\ MyLibrary.dll$$

Esempio 3.10 - Avvio di Pex con le opzioni di default

con queste opzioni di avvio lo strumento esplora tutte le classi nella libreria. È possibile impostare un certo numero di filtri per determinare quali parti software si vogliono esplorare, per esempio sul nome del namespace o sulla classe, se si hanno i sorgenti a

[31] Finestra dei comandi MS-DOS: Anche chiamato il Prompt dei comandi è una funzionalità di Windows che permette la digitazione di comandi MS-DOS (Microsoft Disk Operating System) ed altri per il computer senza utilizzare l'interfaccia grafica di Windows.

disposizione, utilizzando anche caratteri jolly come l'asterisco ∗ per selezionarne un insieme.

Durante l'analisi dello strumento per la scrittura di questo documento, si è avviato Pex nel seguente modo:

```
pex.exe MyLibrary.Tests.dll /nor /ftf
```

Esempio 3.11 - Avvio di Pex da linea di comando partendo dalla libreria di test

utilizzando le opzioni per la creazione della reportistica ed utilizzando i test già salvati nella libreria analizzata, nella cartella di esecuzione sono presenti anche la libreria da testare.

Di seguito un esempio di come Visual Studio avvii Pex con tutte le opzioni impostate:

```
pex.exe "D:\Ett.Common\bin\Debug\Ett.Common.dll"
/assemblyresolutiondirectories:"c:\Program Files (x86)
\Microsoft Visual Studio 10.0\Common7\IDE\PublicAssemblies"
/explorationreflectionmode:LazyWizard
/instrumentedassemblies:log4net;System.configuration;System.ServiceProcess
/x64failsilently /clrversion:v4.0.30319 /donotopenreport
/reportrootpath:"D:\Ett.Common\bin\Debug\reports"
/testassemblyname:Ett.Common.Tests
/testframework:VisualStudioUnitTest /testlanguage:cs
/testprojectfile:"D:\Ett.Common\Ett.Common.csproj"
/testprojectnotupdate /testprojectskip
```

Esempio 3.12 - Comandi di avvio di Pex utilizzato da Visual Studio

Infine di seguito un altro esempio per la verifica della libreria del solver Z3 per cui non si dispongono i sorgenti, libreria utilizzata da Pex (va precisato che in questo caso è necessario avviare la finestra dei comandi MS-DOS con un utente amministratore) .

```
Amministratore: C:\Windows\System32\cmd.exe - "C:\Program Files\Microsoft Pex\bin\pex.x86.exe...
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Windows\system32>"C:\Program Files\Microsoft Pex\bin\pex.x86.exe"  "C:\Progra
m Files\Microsoft Pex\bin\Microsoft.Z3.dll" /nor /ftf /erm:wizard
Microsoft Pex v0.94.51023.0 -- http://research.microsoft.com/pex -- v4.0.30319
Copyright (c) Microsoft Corporation 2007-2010. All rights reserved.

[tests] could not resolve test framework, using default (Visual Studio Unit Test
)
instrumenting... launched Pex 0.94.51023.0 x86 Edition on .NET v4.0.30319
[tests] could not resolve test framework, using default (Visual Studio Unit Test
)
00:00:00.0> starting execution
  00:00:00.0> reflecting tests
    [symbols] could not load symbols for C:\Windows\assembly\GAC_32\Microsoft.Z3
\2.0.40217.6__31bf3856ad364e35\Microsoft.Z3.dll
    [symbols] search path: ;
    [tests] symbol test hasher not supported: could not find symbols for C:\Wind
ows\assembly\GAC_32\Microsoft.Z3\2.0.40217.6__31bf3856ad364e35\Microsoft.Z3.dll
    [tests] testhasher Microsoft.Pex.Engine.TestGeneration.PexSymbolTestHasher n
ot supported
  00:00:03.8> Microsoft.Z3
    00:00:03.9> AstTest
      00:00:03.9> CompareTo(Ast, Object)
          !warning! [explorables] could not guess how to create Microsoft.Z3.Ast

        !warning! [execution] could not generate any test in 0 runs
      00:00:05.2> Equals01(Ast, Object)
          !warning! [explorables] could not guess how to create Microsoft.Z3.Ast

      00:00:05.3> GetHashCode01(Ast)
          !warning! [explorables] could not guess how to create Microsoft.Z3.Ast

      00:00:05.3> ToString01(Ast)
          !warning! [explorables] could not guess how to create Microsoft.Z3.Ast

    00:00:05.3> ConfigTest
      00:00:05.3> Constructor()
        [test] (run 1) Constructor420 (new)
        [coverage] coverage increased from 0 to 3 blocks (+3) after flipping (un
known method) at 0x(unknown offset)
        [dynamic coverage] 3/3 block (100,00%)
      00:00:06.1> Dispose(Config)
          !warning! [explorables] guessed how to create Microsoft.Z3.Config
```

Esempio 3.13 - Avvio di Pex su una libreria di cui non si dispone del codice sorgente

3.2.4.3. Reportistica

Ad ogni esecuzione di Pex, salvo configurazioni di avvio particolari, viene creata una cartella chiamata in base all'ora di avvio dell'esecuzione nel seguente formato:

"$AnnoMeseGiorno.OreMinutiSecondo.Numero.pex$"

($Esempio\ 121110.175630.1800.pex$)

Dove all'interno viene creato un sito web con il dettaglio dell'esplorazione.

La pagina da aprire con un browser per poter visitare il sito web è "$report.html$" . Il titolo mette in evidenza le statistiche generali, viene riportato il numero di esplorazioni

effettuate, il numero di test generati, eventuali eccezioni generate ed il tempo totale impiegato per l'esplorazione.

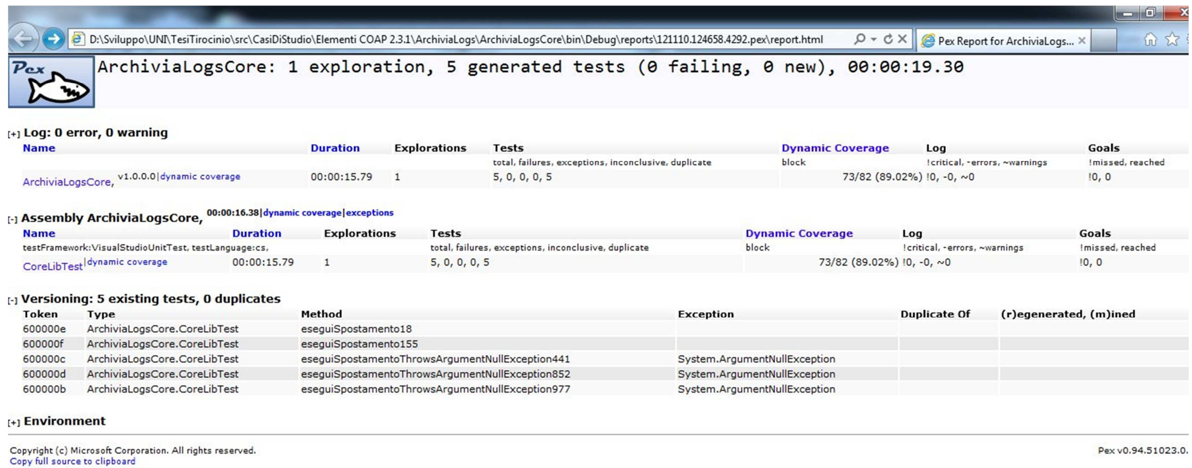


Figura 3.6 - Sito web del report prodotto dall'esplorazione

Navigando nelle pagine si possono scoprire :

- Tutte le classi e metodi visitati, cliccando il collegamento "*Dynamic Coverage*" (letteralmente "copertura dinamica") è possibile accedere alle percentuali di copertura raggiunta per ogni classe;
- Tutte le PUT generate e relativi valori di input utilizzati;
- Se i file sorgenti sono a disposizione, viene creata una pagina web per ogni test e vengono colorate le righe di codice del file sorgente in base alla copertura raggiunta, diversamente si avranno solo le percentuali ed i nomi dei metodi testati.

Oltre al sito web viene anche creato un file in formato XML chiamato "*report.per*" . Esso potrebbe essere importato dalla finestra di Visual Studio "*Pex Exploration results*" per poter analizzare l'esplorazione. Infine viene anche creato un file "*output.txt*" con i messaggi ed eventuali errori generati nell'esecuzione .

La creazione del sito di reportistica non è attiva in Visual Studio, è necessario abilitarla dalla finestra delle "*opzioni personali*" dedicata a Pex.

3.2.4.4. Documentazione e supporto a disposizione

Per chiunque volesse approfondire la conoscenza di questo strumento, sono a disposizione diverse fonti di informazione.

Dentro la cartella di installazione è presente la sottocartella “ $Documentation$ " (letteralmente documentazione) contenente un insieme di *tutorial* (letteralmente guida) associati ad un insieme di progetti di esempi con il codice sorgente ed archiviati nel file " $pex.samples.zip$ ". Nella cartella c'è anche la documentazione ufficiale per gli sviluppatori, "Microsoft Pex SDK Documentation" fornita nella forma di "Microsoft Compressed HTML Help" [32] file " $pex.documentation.chm$ ". È anche presente una ulteriore cartella "$wiki$" contenente la documentazione in forma di sito web.

Ulteriore documentazione è disponibile sul sito web ufficiale [2], sono presenti documenti in formato Microsoft World e file in formato PDF [33]. Pex, essendo nato come progetto di ricerca, ha fatto si che uscissero molte pubblicazioni che parlino di esso.

Infine attraverso il sito web *stackoverflow.com* [28] è possibile chiedere supporto sotto forma di domanda o consultare le risposte di quelle già presenti.

[32] Microsoft Compressed HTML Help: Letteralmente “aiuto in HTML compresso”, è un formato di guida con indici e finestra di ricerca creato da Microsoft per la creazione di documentazione.
[33] PDF: “Portable Document Format”, letteralmente “formato di documento portabile” è un formato di documento creato con un linguaggio di programmazione al suo interno, pensato come formato universale per la diffusione di documenti.

CAPITOLO 4 Strumenti per la generazione ed esecuzione automatica di Test funzionali per applicazioni web

Nell'ultimo decennio le applicazioni software hanno avuto una notevole evoluzione, da applicazioni strettamente legate al computer dove venivano utilizzate, ad applicazioni basate su architetture di rete distribuite, attraverso l'utilizzo di pagine web, grazie anche all'espansione della rete Internet. La tecnologia web, è il modo in cui vengono visualizzati ed interpretati documenti scritti in formato HTML, realizzabili con diversi linguaggi di programmazione per renderli dinamici ed interattivi con il lettore. Quest'ultimo non sa come sono state realizzate le pagine di un sito web e per questo motivo si preferisce applicare test di tipo Black-Box Testing, argomento trattato nel paragrafo 2.4.2.1 - Black-Box Testing, conoscendo invece il risultato aspettato.

Di seguito lo studio di strumenti e concetti per l'esecuzione di test funzionali.

4.1. Strumenti esistenti

Effettuando una ricerca, per trovare sia applicazioni software che framework per lo sviluppo di test di verifica funzionale, sono emersi diversi strumenti.

Selenium automates browsers

Nato [29] inizialmente come strumento sviluppato in Java [30], noto linguaggio di programmazione ad oggetti, e con l'utilizzo del solo browser Firefox [31], lo strumento è diventata una suite software composta dai seguenti moduli:

- *Selenium IDE*: estensione del noto browser Firefox, permette la registrazione della visita di un sito e la possibilità di replicare la navigazione eseguita;
- *Selenium Remote Control*: è un sistema client/server che permette il controllo della navigazione attraverso l'utilizzo di un browser locale o remoto attraverso l'utilizzo di un linguaggio di programmazione creando unit test. All'atto della scrittura di questo documento i linguaggi supportati sono tutti i principali linguaggi ad oggetti;

- *Selenium WebDriver*: sono moduli aggiuntivi per il supporto di browser differenti. All'atto della scrittura di questo documento sono supportati tutti i principali browser attualmente esistenti;
- *Selenium Grid*: permette il controllo di più navigazioni contemporanee governando istanze diverse del modulo "*Selenium Remote Control*".

Tutti i software Selenium hanno licenza "Apache 2.0 License" [32] ad uso gratuito, la società produttrice offre supporto e consulenza per progetti particolari su richiesta.

WatiN

Pronunciato [3] what-in, con il significato di “Web Application Testing In .Net” (letteralmente “Testing di applicazioni web con .NET”), è un framework per sviluppatori C# per la navigazione e manipolazione delle pagine web navigate in tempo reale per poter effettuare testing automatico delle applicazioni web.

È stato scelto ed utilizzato per lo sviluppo di uno strumento creato per poter effettuare testing automatico, argomento affrontato nei successivi paragrafi.

Sahi

Lo strumento [33] permette, attraverso uno script di configurazione, di lanciare la navigazione di siti web ed avere un report in diversi formati. È possibile registrare una navigazione ed effettuarla nuovamente partendo dallo script generato in precedenza.

Esistono due versioni, una open source e l'altra a pagamento (Sashi Pro) per il costo di circa 495 $ (Dollari americani), esse si differiscono per moltissime funzionalità. La versione open permette soltanto la registrazione ed esecuzione di navigazioni, quella a pagamento permette la creazione degli script attraverso un programma di sviluppo. Quest’ultima supporta anche le pagine in Flex[34] [34] e permette il salvataggio dei risultati su database.

La versione open source è rilasciata con licenza "Apache 2.0 License" ad uso gratuito.

[34] Adobe Flex : Linguaggio proprietario di Adobe per la creazioni di animazioni integrate in pagine web. È necessario utilizzare una estensione del browser per poterle visualizzare.

SoapUI

Lo strumento [35] è una applicazione Java inizialmente nata per poter verificare il funzionamento di "web service" (letteralmente "servizi web"), oggi è utilizzata per poter testare protocolli anche diversi da SOAP come REST o siti attraverso HTTP [35].

Oltre a dare la possibilità di effettuare test funzionali attraverso la registrazione dell'invocazione di un servizio o applicazione web, lo strumento è orientato ad effettuare test sulla sicurezza e sulla gestione delle risorse della applicazione testata. Esistono due versioni, una gratuita "SoapUI OS" e l'altra a pagamento "SoapUI Pro" per il costo annuale che varia da 313 a 349 $ (Dollari americani) in base alla durata (da uno a tre anni) della licenza che si vuole comprare. Tra le due versioni le differenze sono notevoli, in quella professionale è possibile avere della reportistica in diversi formati, un generatore casuale di input ed il supporto del produttore via posta elettronica.

4.2. WatiN, libreria C# per l'automazione della navigazione di applicazioni web

Tra i vari strumenti, si è scelto l'analisi nel dettaglio ed il completo utilizzo di WatiN per la sua semplicità di utilizzo ed apprendimento.

L'idea di base è stata quella di scegliere una libreria per il linguaggio C# per poter sviluppare un nuovo software come strumento di testing utile per l'azienda ETT s.r.l.

4.2.1. Caratteristiche

Al momento della scrittura di questo documento è possibile utilizzare la versione 2.1.0.1196, rilasciata il 12 aprile 2011. Dalla versione 1.0.0.400, il software è rilasciato con la licenza "Apache License 2.0" [32], il progetto è open source e chiunque può contribuire agli sviluppi.

---

[35] SOAP, REST, HTTP: Essi sono protocolli di rete per la comunicazione tra servizi web. SOAP acronimo di "Simple Object Access Protocol" (letteralmente "protocollo per l'accesso a semplici oggetti") è un protocollo per lo scambio di messaggi tra componenti software, spesso i componenti sono su reti distribuite. REST ("REpresentational State Transfer", letteralmente "rappresentazione dello stato traferito") ed HTTP ("HyperText Transfer Protocol", letteralmente "protocollo di trasferimento di testo dinamico") sono protocolli orientati alla creazione degli indirizzi per accedere alle risorse web, il primo più complesso con la rappresentazione del dominio dello stato applicativo, il secondo considerato solo come riferimento.

Di seguito le principali caratteristiche.

- Supporta i framework .Net 2.0, 3.5 e 4.0;
- È compatibile con tutte le versioni di Internet Explorer fino alla versione 9, per quest'ultima, inizialmente erano presenti alcune problematiche che sono state corrette nell'ultimo rilascio;
- È possibile utilizzare il browser Firefox [31] fino alla versione 3.6 dopo aver installato l’estensione "jssh" inclusa nel progetto di WatiN. Essa è utilizzato da WatiN per automatizzare l'utilizzo di FireFox;
- È stata predisposta una classe sperimentale $Watin.Core.Native.ChromeBrowser$ per l'utilizzo del browser Chrome [36] ma di fatto non è implementata e quindi non è supportato;
- È possibile istanziare oggetti di tipo browser e attraverso il linguaggio C# pilotare una navigazione di una o più applicazione web;
- Durante la navigazione si possono manipolare gli oggetti HTML delle pagine navigate;
- È possibile creare unit test navigando nelle pagine web;
- Attraverso la sintassi dei selettori CSS[36] [37] è possibile intercettare gli elementi, anche complessi JavaScript come ad esempio un campo di tipo data rappresentato con l’elemento *DatePicker* ;
- Utilizzando un software sviluppato esternamente al progetto "WatiN Test recorder" [38] (letteralmente “Registratore di Test per WatiN”) è possibile registrare le navigazioni, generare del codice in linguaggio C# ed utilizzarlo nei propri progetti di sviluppo;
- Supporta JavaScript e chiamate AJAX[37];
- Si possono cliccare bottoni e collegamenti testuali avviando la navigazione in una nuova pagina.

4.2.2. L’utilizzo

[36] CSS: Il “Cascading Style Sheets”, letteralmente “Fogli di stile a cascata” è un linguaggio per la definizione di come formattare gli oggetti un una pagina HTML. Si definiscono colori, dimensioni e stili dei singoli oggetti. La ricerca dell’oggetto avviene attraverso un selettore.

[37] AJAX: “Asynchronous JavaScript And XML”, è una tecnica di sviluppo per realizzare applicazioni web interattive dividendo la parte di controllo eseguita in JavaScript e la parte dati, modellata con oggetti XML

Non vi è un pacchetto di installazione per utilizzare WatiN, è necessario, dopo aver scaricato l'applicativo dal sito ufficiale [3], scompattarlo in una cartella.

Durante lo sviluppo dei propri progetti, sarà possibile referenziare la libreria $WatiN.Core.dll$, disponibile nella cartella "*bin*" nella cartella di installazione.

4.2.2.1. Configurazioni per l'avvio

A causa del modo di funzionamento di Internet Explorer e l'utilizzo della libreria di tipo COM ("Common Language Runtime", letteralmente "linguaggio comune disponibile in tempo reale") la situazione di interoperabilità comporta l'utilizzo di WatiN in modalità *Thread Apartmentstate* uguale a STA ("Single Thread Appartment", significa "modalità singolo thread"). Le librerie COM utilizzano gli *apartment*[38] dove è necessario definire la modalità di accesso nel momento dell'utilizzo. L'impostazione STA, definisce l'accesso di un solo thread alla volta. È anche possibile utilizzare l'impostazione MTA ("Multithreaded Apartments", significa "modalità con più di un thread") per effettuare l'accesso con più thread contemporaneamente. In .NET framework non vengono utilizzati gli apartment e gli oggetti sono responsabili di utilizzare tutte le risorse condivise in modalità *thread-safe* [39]. Poiché Internet Explorer non è *thread-safe* è necessario accedere alla libreria di interoperabilità in modalità STA.

Per procedere nello sviluppo, nel proprio codice sorgente è necessario:

1. Referenziare la libreria $WatiN.Core.dll$;
2. Applicare l'attributo $[STAThread]$ al metodo principale (il punto di ingresso) del programma di tipo *Console Windows* o *Windows Form*.[40]

```
[STAThread]
static void Main (string [] args)
```

Esempio 4.1 - Applicazione attributo STA

[38] Apartment: Letteralmente appartamento ma parlando di sviluppo Microsoft è un contenitore logico all'interno di un processo per gli oggetti che condividono gli stessi requisiti di accesso di un thread. Tutti gli oggetti nello stesso apartment possono ricevere le chiamate da qualsiasi altro thread nell'apartment.

[39] Thread-safe: Modalità dove l'accesso alle risorse da parte dei thread coinvolti avviene in modo sincronizzato accedendo in modo sicuro senza la preoccupazione che altri processi utilizzino la risorsa.

[40] Console Widows,Windows Form: Sono due tipologie di applicazioni in Microsoft .NET, la prima di tipo testuale, la seconda grafica con le finestre (le *Form*) .

È possibile utilizzare WatiN per creare delle unit test sia con il framework NUnit [11] che con MbUnit [39] . Nel caso di NUnit è necessario creare un file di configurazione per impostare la modalità di accesso alla libreria COM per utilizzare Internet Explorer nella navigazione.

Se si utilizza NUnit in un progetto di Visual Studio basterà aggiungere la seguente proprietà nel file di configurazione.

$$< add = "ApartmentState" \ value = "STA" />$$

Di seguito un esempio di file di configurazione.

```
<? Xml version = codifica "1.0" = "utf − 8"? >
< configuration >
< configSections >
< sectionGroup name = "NUnit" >
 < section name = "TestRunner" type
                 = "System.Configuration.NameValueSectionHandler"/>
</ SectionGroup >
</ ConfigSections >
< NUnit >
< TestRunner >
< ! − − I valori validi sono STA, MTA. Altri ignorato. − −>
< add key = "ApartmentState" value = "STA" />
</ TestRunner >
</ NUnit >
</ Configuration >
```

Esempio 4.2 - Esempio di file di configurazione per impostare STA in NUnit

Dalle ultime versioni del framework NUnit è anche possibile utilizzare l'attributo [*RequiresSTA*] al posto dell'impostazione dal file di configurazione.

Di seguito un modello di esempio di classe di testing con le unit test da sviluppare.

```
[TestFixture]
[RequiresSTA]
public class NomeClasseTest
{
...
private Browser _browser;

[SetUp]
public void Inizializza()
{
_browser = new IE();
}

[TearDown]
public void Dispose()
{
if(_browser != null)
_browser.Dispose();
}

[Test]
public void UnitTest1()
{
...
_browser.*
Assert.*
...
}
```

```
[Test]
public void UnitTest2()
{
...
_browser.*
Assert.*
...
}
...
}
```

Esempio 4.3 - Esempio di classe di testing con NUnit e WatiN

4.2.2.2. Le librerie

Le librerie di WatiN sono divise in diversi namespace, di seguito una breve spiegazione.

- $Watin.Core$: è il motore di WatiN, qui si possono trovare tutti gli elementi HTML modellati;

Tipologia	**Tag HTML**	**Classe WatiN Singolo Elemento**	**Classe WatiN Lista di Elementi**	**Esempio**
Collegamento	<a />	Link		var link = browser.Link(linkId);
Area di testo	<area />	Area		var area = browser.Area(Find.ByAlt(alttext));
Pulsante	<button />	Button		var button = browser.Button(buttonId);
Divisione	<div />	Div		var div = browser.Div(divId);
Modulo	<form />	Form		var form = browser.Form(formId);
Frammento HTML	<frame />	Frame		var frame = browser.Frame(frameId);
Insieme di Frammenti	<frameset />	FrameCollection		var frames = browser.Frames;
Frammento nei frammenti	<iframe />	Frame		var frame = browser.Frame(frameId);
Immagine	<img />	Image		var image = browser.Image(imageId);
Pulsante	<input type=button/>	Button		var button = browser.Button(buttonId);
Spunta	<input type=checkbox/>	CheckBox		var checkbox = browser.CheckBox(checkboxId);
File	<input type=file/>	FileUpload		var upload =browser.FileUpload(fileuploadId);
Campo nascosto	<input type=hidden/>	TextField		var textfield = browser.TextField(hiddenId);

Immagine	<input type=image/>	Image		var image = browser.Image(imageId);
Campo password	<input type=password/>	TextField	TextFieldCollection	var pass = Ie.TextField(passwordId);
Opzione	<input type=radio/>	RadioButton	RadioButtonCollection	var radio = Ie.RadioButton(radioId);
Pulsante reset modulo	<input type=reset/>	Button	ButtonCollection	var reset = Ie.Button(resetId);
Campo di testo	<input type=text/>	TextField	TextFieldCollection	var text = Ie.TextField(textId);
Etichetta	<label />	Label	LabelCollection	var label = Ie.Label(elementId);
Opzione di scelta	<option />	Option	OptionCollection	Var opt = Ie.Select(selectId).Options;
Paragrafo	<p />	Para	ParaCollection	var para = Ie.Para(pId);
Selezione	<select />	Select	SelectCollection	var select = Ie.Select(selectId);
Spaziatura	<span />	Span	SpanCollection	var span = Ie.Span(spanId);
Tabella	<table />	Table	TableCollection	var tb = Ie.Table(tableId);
Corpo della tabella	<tbody />	TableBody	TableBodyCollection	var tbody = Ie.TableBody(tablebodyId); var tbodies = Ie.Table(tableid).TableBodies;
Cella della tabella	<td />	TableCell	TableCellCollection	var cell = Ie.TableCell(tablecellId); var cell = Ie.Table(TableId) .TableRows[0].TableCells[0];
Area di testo	<textarea />	TextField	TextFieldCollection	var area = Ie.TextField(textareaId);
Riga nella tabella	<tr />	TableRow	TableRows	var row = Ie.TableRow(tablerowId); var row = Ie.Table(TableId).TableRows[0];
Qualsiasi elemento HTML		Element ElementsContainer	ElementCollection	var elem = Ie.Element(elementId); Var elem = Ie.Element(tagname, elementId);

Tabella 4.1 - Elementi HTML supportati da WatiN

- *Watin.Core.Constrains*: in questo namespace sono presenti diverse classi che rappresentano i criteri di ricerca utilizzati per cercare gli elementi HTML nella pagina aperta, che si vogliono utilizzare;
- *Watin.Core.DialogHandlers*: per poter intercettare e controllare le finestre di dialogo, WatiN mette a disposizione questo insieme di classi;
- *Watin.Core.Interfaces*: le interfacce definite in questo namespace, permettono l'implementazione di nuove funzionalità. Per esempio un nuovo browser dovrà almeno implementare l'interfaccia *ISettings* per definire tutte le configurazioni utilizzate poi da WatiN durante la navigazione;
- *Watin.Core.Logging*: libreria per poter utilizzare i messaggi informativi prodotti dalla libreria stessa;
- *Watin.Core.Native*: qui si possono trovare tutte le classi per la navigazione, per esempio tutte le implementazioni dei browser supportati.

La struttura è semplice ed intuitiva i namespace *Watin.Core.Constrains* ed *Watin.Core.Native* sono normalmente le più utilizzate.

4.2.2.3. Classiche implementazioni

Normalmente le applicazioni che utilizzano WatiN vengono sviluppate attraverso i seguenti passi:

1. Si inizializza l'oggetto $WatiN.Core.Browser$, istanziando la classe del tipo di browser che si vuole utilizzare (esempio per Internet Explorer la classe $WatiN.Core.IE$);
2. Si apre una pagina web attraverso il metodo $Browser.GoTo$, si cercano gli elementi HTML della pagina interessati e si istanziamo attraverso le classi define (menzionate nella Tabella 4.1 - Elementi HTML supportati da WatiN);
3. Si generano azioni sugli oggetti istanziati come clic, valorizzazioni o selezioni, generando un cambiamento dell'oggetto browser che rappresenta la pagina visitata. Si potrebbe ritornare al passo 2;
4. Attraverso alcuni metodi messi a disposizione per la ricerca di testo o di elementi nella pagina visitata è possibile definire degli *Assert* per la verifica del contenuto. Dopo la verifica si potrebbe terminare o proseguire dal passo 2 o 3;
5. Quando si decide di terminare la navigazione è buona norma effettuare la chiusura del browser attraverso il metodo $Browser.Close$.

Algoritmo 4.1 - Esempio di passi da svolgere per lo sviluppo con WatiN

WatiN è stato pensato principalmente per lo sviluppo di unit test utilizzando il framework NUnit, ma come evidenziato, è anche possibile effettuare semplici navigazioni per sollecitare eventi automatici in applicativi di tipo web.

4.2.3. Documentazione

Il progetto è ben documentato, la documentazione ufficiale con le API della libreria si possono trovare nel documento $WatiN.chm$ nella cartella dell'installazione.

Dal sito ufficiale esiste un'area dedicata composta da un insieme di guide dove tutte trattano un breve argomento oltre, ad una pagina di domande e risposte per poter iniziare a sviluppare.

Esiste anche la possibilità di iscriversi ad una mailinglist[41] per poter chiedere supporto via email alle persone che già utilizzano WatiN. Anche sul sito web *stackoverflow.com* [28] è possibile chiedere supporto sotto forma di domanda o consultare le risposte di quelle già fatte.

Infine è stato creato un sito blog "WatiN and more" [40] (letteralmente "WatiN e di più") creato da Jeroen van Menen, creatore del progetto, dove si possono trovare articoli correlati.

4.2.4. Conclusioni

L'impressione dello strumento nel suo utilizzo, è stata come da aspettative, è stato semplice da imparare, ma la più grande limitazione è sicuramente il numero di browser supportato al quanto limitato.

Il requisito fondamentale era l'utilizzo di Internet Explorer 9 e solo in un caso si è rilevato che qualche funzionalità JavaScript non era ben supportata (il messaggio informativo di invocato con il metodo $Alert$ non si è riusciti a gestirlo), da una prima analisi è emerso che poteva essere una impostazione nell'utilizzo del browser da correggere.

Tutti gli oggetti HTML sono supportati e non sono mai stati rilevate anomalie particolari dimostrando la sua stabilità.

Un punto a favore lo ha giocato la classe generica $Watin.Core.Element$, per la rappresentazione in quanto frequentemente utilizzata dell'analisi della pagina visitata insieme alla ricerca per selezione utilizzando la sintassi dei selettori CSS particolarmente pratica.

Non si è trovato il modo di effettuare chiamate POST o SOAP per il sollecito di servizi web, attraverso il metodo $Browser.GoTo$ è stato possibile solo effettuare chiamate di tipo GET[42].

[41] Mailinglist : Letteralmente "lista di poste elettroniche". Chi è iscritto può scrivere una email alla mailing list, la quale la riceveranno tutte le persone che ne fanno parte, che si saranno iscritti con un indirizzo di posta elettronica.
[42] POST , GET : Sono i due metodi supportati dal protocollo HTTP per richiedere risorse web .

L'ultimo rilascio di WatiN risale al 2011, la comunità è ancora attiva ma l'impressione è quella che il progetto non sia più portato avanti. Nei blog degli sviluppatori non vi sono articoli dedicati recenti. La controparte è che c'è ancora molto interesse dimostrato dai numerosi messaggi nel forum ufficiale, sia per la richiesta di nuove funzionalità che per la richiesta di supporto nell'utilizzo.

4.3. Strumento sviluppato per la generazione dei dati di input per il Testing di pagine web utilizzando la libreria WatiN

L'importanza di effettuare test funzionali, prima della messa in produzione di un progetto software, ha portato alla necessità di pensare alla progettazione e sviluppo di un software nuovo, focalizzato sul testing automatico di applicazioni web attraverso l'utilizzo della libreria WatiN. Con l'idea che fosse anche un generatore dei dati di input, è stato chiamato "*Automatic Tester of Web Pages 1.0*" .

4.3.1. Introduzione

Partendo dall'analisi di ciò che un tester svolge quotidianamente per testare un sito web, si sono definiti alcuni obbiettivi che lo strumento, una volta finito lo sviluppo, dovrà andare a soddisfare.

Gli obbiettivi che si sono preposti sono stati :

- Navigare nelle applicazioni web e poter verificare il contenuto delle pagine visitate;
- Creare un generatore di input per poter compilare ed inviare moduli web;
- Poter definire il dominio dei dati di input da generare;
- Permettere la generazione dei dati di input sia dentro che fuori il dominio definito;
- Poter verificare il contenuto delle nuove pagine generate in funzione dei dati forniti;
- Poter effettuare gli stessi test più volte;
- A fine esecuzione avere un report sui test effettuati;
- Partendo da una pagina, visitare tutte quelle ad essa collegate.

Di seguito alcune funzionalità sottointese ma non banali e da ritenersi fondamentali.

- Supporto dell'interpretazione di funzioni JavaScript;
- Completo supporto del browser Internet Explorer 9;
- Linguaggio di programmazione C#.

Queste ultime sono state considerate supportate nel momento in cui si è scelto WatiN come libreria adottata per lo sviluppo.

4.3.2. Funzionalità

L'applicazione parte leggendo un file di configurazione dove è possibile specificare come effettuare i test.

La principale funzionalità dello strumento è quella di testare il funzionamento di una pagina web. Il test potrà essere di due tipi :

- *FORMPAGE*: per compilare un modulo (spesso chiamato *form*) e verificare il suo invio;
- *LINKPAGE*: per aprire una pagina web e verificare il nuovo contenuto;

Nel caso della compilazione di un modulo, si sono scelti diversi criteri di selezione degli input, in particolare si sono scelte cinque modalità di utilizzo:

- *CASUALE_DEFAULT*: i dati di input sono quelli specificati nel campo HTML che si sta compilando;
- *CASUALE_DOMINI*: sapendo il dominio di valori, per ogni campo vengono generati valori casuali appartenenti al dominio;
- *CASUALE_FUORIDOMINIO*: sapendo il dominio dei valori, per ogni campo vengono generati valori casuali, sicuramente non appartenenti al dominio;
- *CASUALE*: vengono generati valori casuali considerando il valore del campo HTML analizzato, un valore dentro ed uno fuori dal dominio;
- *FUNZIONALE*: ad ogni campo si specifica il valore da utilizzare per poter essere certi del risultato atteso e poterlo verificare, la selezione dei dati è compito di chi configura il test.

I criteri di selezione degli input permettono l'esecuzione di test sia strutturali che funzionali, per quelli strutturali si potrà verificare che la pagina non sia in uno stato non gestito dall'applicativo testato, si verifica quindi una funzionalità.

Come definito nel paragrafo 2.4.2.1 - Black-Box Testing il dominio dei valori (IDM) può essere basato sull'interfaccia o basato sulle funzionalità, pertanto si è creato un insieme di possibili domini funzionali.

Tipologia dato o dominio	**Significato**
AZIENDA	Testo con sole lettere
CAP	Codice di avviamento postale
CAPTCHAASPX	Calcolo Codice CAPCHA calcolato secondo il linguaggio aspx
CHECK	Valore vero o falso
CODICEFISCALEA	Codice fiscale aziendale
CODICEFISCALEP	Codice fiscale per persona fisica
COGNOME	Testo con sole lettere
COGNOMECFP	Cognome appartenente ad un codice fiscale
COMUNE	Comune esistente
COMUNECF	Comune di nascita riferito ad un codice fiscale
DATA	Data generica
DATANASCITACF	Data di nascita riferita ad un codice fiscale
EMAIL	Email generica
NODOMINIO	Nessun significato funzionale specificato
NOME	Testo con sole lettere
NOMECFP	Nome appartenente ad un codice fiscale
NUMEROCONVIRGOLA	Numero generico con la virgola
NUMEROINTERO	Numero generico intero
PASSWORD	Testo alfanumerico con qualsiasi carattere
TESTOLIBERO	Testo con sole lettere
URL	Indirizzo sito web
USERNAME	Testo alfanumerico
VALOREHTML	Valore specificato nella pagina web
VALORETEST	Valore specificato nel file di configurazione
VIEWSTATEASPX	Valore del *viewstate* in una pagina aspx
VUOTO	Nessun valore
XML	Testo in formato XML

Tabella 4.2 - Domini dei dati basati sulle funzionalità

Nel caso non sia specificato, utilizzando la configurazione $NODOMINIO$ viene considerato il tipo del campo HTML analizzato (dominio basato sull'interfaccia).

Dopo l'esecuzione del test, verrà effettuata una verifica sul contenuto della pagina discriminando l'esito del test con un OK se la verifica ha avuto esito positivo e KO in caso contrario.

Alla fine di ogni esecuzione verranno generati sempre due file con il nome in funzione dell'ora di fine esecuzione dei test. I file sono creati in formato Microsoft Excel [41], per scelta ogni foglio elettronico rappresenta i dati dei test effettuati per un sito.

In particolare è generato:

- *un file contenente il report con i risultati dei test effettuati:* esso ha sempre il nome nel seguente formato

  $$NOMEPROGETTO_Anno - Mese - Giorno_OreMinutiSecondi.xls$$

  le colonne contenute nel report sono:
  - *Test Id*: numero identificativo del singolo test effettuato;
  - *Nomi dei campi*: se il test prevede la compilazione di un modulo, verranno riportati tutti gli identificativi dei campi trovati con i valori generati ed utilizzati nel test, un campo per colonna;
  - *Testo atteso*: testo che ci si aspetta di trovare nella pagina dopo l'esecuzione del test;
  - *Testo da non trovare*: testo che non ci si aspetta di trovare dopo l'esecuzione del test;
  - *Esito*: esito del test, valorizzato con OK se passato o KO se fallito;
  - *Durata*: durata in secondi dell'esecuzione del test;
  - *Errore*: eventuali errori che hanno fatto fallire l'esecuzione del test.
- *nel caso il test si basi sulla verifica della compilazione di un modulo, verrà generato anche un file contenente tutti i dati utilizzati nei test effettuati*: esso ha sempre il nome nel seguente formato

  $$NOMEPROGETTO_Dati_Anno - Mese - Giorno_OreMinutiSecondi.xls$$

  le colonne contenute sono:
  - *Test Id*: numero identificativo del singolo test effettuato;
  - *Passo Test Id*: numero dello step (letteralmente passo) del singolo test, se vi sono più moduli di seguito come in un flusso, il numero sarà progressivo a discriminare l'ordine del modulo a cui i dati fanno riferimento;
  - *Nome Parametro*: nome univoco del campo nel modulo;
  - *Valore*: valore utilizzato nel test corrente;

- *Dominio*: come precedentemente definito, il dominio funzionale configurato dall'utente ed associato al campo;
- *Testo Trovato/Da Trovare*: Testo cercato nella pagina raggiunta dal test, in funzione del valore del campo.

Quest'ultimo file è possibile utilizzarlo come file dei dati per l'esecuzione dei test già eseguiti per verificare se il sistema continua a comportarsi come si era comportato in precedenza.

Ad ogni esecuzione, lo strumento crea o modifica un file di log nella stessa cartella del programma (file $GenerateInputWeb.log$).

4.3.3. File di configurazione

Dal file di configurazione si definisce il comportamento dello strumento che dovrà avere alla sua esecuzione.

Creato con una struttura in formato XML si è utilizzata la tecnica di sviluppo "Custom Section Configuration" [42] (letteralmente "sezione di configurazione personalizzata"), sfruttando il file di configurazione dell'applicazione oppure un file esterno con una sezione dedicata alla configurazione dei test da eseguire.

Da esso è possibile definire le seguenti proprietà di esecuzione:

- Dati di personalizzazione del progetto di test, nome del progetto, la cartella dove creare i file della reportistica e dei dati ed eventualmente un file dei dati per i test funzionali o di regressione;
- Un elenco di browser per eseguire i test;
- Un elenco di siti da testare, specificando la modalità per la generazione degli input definiti nel precedente paragrafo;
- Per ogni sito è possibile definire un insieme di pagine su cui effettuare i test e le specifiche configurazioni. È stato pensato un insieme di pagine poiché la stessa pagina web potrebbe esistere in realtà diverse, raggiungibile da diversi indirizzi;
- Per ogni pagina è possibile definire un insieme di passi preliminari ed un insieme di passi finale, per raggiungere la pagina del test e per concludere la sessione di test.

L'esempio più classico è testare un'area privata di una applicazione web e quindi effettuare l'accesso prima del test ed uscirne dopo. Ogni passo definisce una azione che potrà essere l'apertura di una pagina, il click di un collegamento o la compilazione di un form specificando i valori da inserire ed il pulsante da cliccare;

- Un insieme di configurazioni di test su ogni sito, permette di verificare funzionalità differenti. Per ogni test si potrà definire la tipologia di pagina, se un modulo da compilare oppure una pagina da aprire (come definito nel precedente paragrafo), un numero massimo di prove da creare, se considerare i campi nel modulo nascosti e se considerare quelli in sola letture. Inoltre si presuppone che il test sia composto da passi evolutivi, per esempio la visualizzazione in sequenza di più moduli o collegamenti uno di seguito all'altro. Per ogni passo si potrà definire l'azione da compiere, un insieme di elementi da trovare ed eventualmente la funzione di generazione dei valori per ogni singolo elemento (o campo del modulo) da utilizzare;
- È possibile configurare un testo che si vuole trovare oppure uno che non si vuole trovare alla fine del test per la validazione di esso.

```xml
<?xml version="1.0"?>
<configuration>
    <configSections>
        <section name="TestProject.Settings" type="GenerateInputWebLib.Model.Configuration.ConfigurationTestProject, GenerateInputWebLib" />
    </configSections>
    <TestProject.Settings Progetto="02 - COL - UniUrg" ReportDir=
"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio\GenerateInputWeb\ConfigurationDist\CO\Rep">
        <DefinizioneTests>
            <Browsers>
                <Browser Valore="IE" />
            </Browsers>
            <Siti>
                <Sito TipologaTest="CASUALE">
                    <Pagine>
                        <Pagina>
                            <FasePreliminare>
                                <Step>
                                    <NavigazioneAction NomeNavigazione = "UNIURG CO" UrlIniziale = "http://w-web01/CO/" />
                                </Step>
                                <Step>
                                    <ContainerElementAction  FiltroPerId = "frmLogin" />
                                    <Elementi>
                                        <ElementoInput FiltroPerId = "txtUtente" >
                                            <ValoriInput>
                                                <Parametro Valore = "nuovasomm" />
                                            </ValoriInput>
                                            <MetodoGeneraInput Metodo="FixedValue" />
                                        </ElementoInput>
                                        <ElementoInput FiltroPerId = "txtPassword" >
                                            <ValoriInput>
                                                <Parametro Valore = "XXXX" />
                                            </ValoriInput>
                                            <MetodoGeneraInput Metodo="FixedValue" />
                                        </ElementoInput>
                                    </Elementi>
                                    <NavigazioneAction FiltroPerNome = "ButtonAccesso" />
                                </Step>
                            </FasePreliminare>
                            <NavigazioneAction NomeNavigazione = "Pagina UNIURG" UrlIniziale =
"http://w-web01/CO/documento.aspx?modulo=4&amp;tipo=1&amp;azione=1" />
                            <FaseFinale>
                                <Step>
                                    <NavigazioneAction NomeNavigazione = "Logout" FiltroPerId = "logout" />
                                </Step>
                            </FaseFinale>
                        </Pagina>
                    </Pagine>
                    <ConfigurationTests>
                        <ConfigTest TipoTest="FORMPAGE"  MaxNumTests="300" ConsideraCampiSolaLettura="false" >
                            <StepsTest>
                                <Step>
                                    <ContainerElementAction  FiltroPerNome = "form1" />
                                    <Elementi>
                                        <ElementoInput FiltroPerId = "Lavoratori txtCodiceFiscale textBox" >
                                            <MetodoGeneraInput Metodo="CFPersonaGenerate" >
                                                <Parametri>
                                                    <Parametro Valore = "1" />
                                                </Parametri>
                                            </MetodoGeneraInput>
                                        </ElementoInput>
                                        <ElementoInput FiltroPerId = "Lavoratori_txtCognome_textBox" DominioValori = "COGNOMECFP" />
                                        <ElementoInput FiltroPerId = "Lavoratori_txtNome_textBox" DominioValori = "NOMECFP" />
                                        <ElementoInput FiltroPerId = "Inizio_txtDataInizio_textBox" >
                                            <MetodoGeneraInput Metodo="DateGenerate" >
                                                <Parametri>
                                                    <Parametro Valore = "1" />
                                                    <Parametro Valore = "dd/MM/yyyy" />
                                                </Parametri>
                                            </MetodoGeneraInput>
                                        </ElementoInput>
                                        <ElementoInput FiltroPerId = "DatiInvio_txtCodiceFiscale_textBox" DominioValori = "CODICEFISCALEA" />
                                    </Elementi>
                                    <NavigazioneAction FiltroPerId = "butInvia" />
                                </Step>
                            </StepsTest>
                            <RisultAspKO ContenutoPagina = "Error" />
                            <RisultAspOK ContenutoPagina = "Comunicazione inviata con successo." />
                        </ConfigTest>
                    </ConfigurationTests>
                </Sito>
            </Siti>
        </DefinizioneTests>
    </TestProject.Settings>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
    </startup>
</configuration>
```

Figura 4.1 - Esempio del file di configurazione

Il file è stato sviluppato in modo incrementale secondo le esigenze che si presentavano, si pensa che in futuro potrà esserci uno strumento esterno per la creazione del file ed avvio del tester automatico.

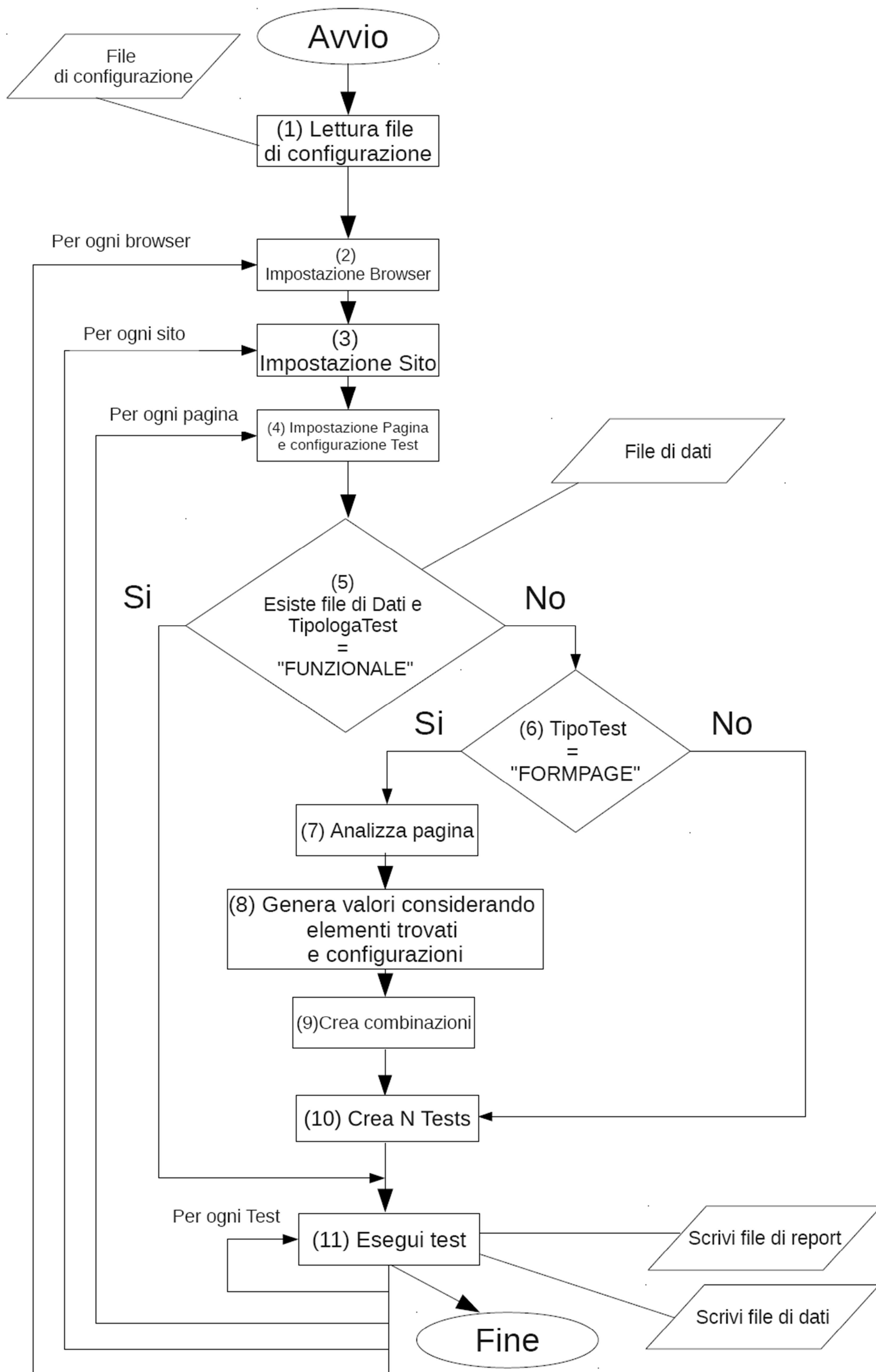
4.3.4. Algoritmo di esecuzione

Avviato lo strumento, esso esegue uno schema ben preciso di operazioni di seguito descritte.

1. *Lettura del file di configurazione*: viene letto e caricato in memoria il file di configurazione;
2. *Impostazione browser*: leggendo la lista di browser configurati, viene istanziato un browser alla volta per poter eseguire le operazioni successive;
3. *Impostazione sito*: leggendo la lista di siti configurati, viene impostato un sito alla volta per poter eseguire le operazioni successive;
4. *Impostazione pagina e configurazione dei test*: considerando che per tutte le pagine impostate, si utilizzeranno le stesse configurazioni di test definite, in questa fase si imposta in memoria la pagina corrente da visitare su cui effettuare i test;
5. *Verifica della tipologia di test del sito*: in questa operazione, si verifica se è stato predisposto un file dei dati e se il test del sito è funzionale (configurazione $TipologaTest = "FUNZIONALE"$). In caso affermativo i test sono definiti dalla lettura del file dei dati e non è necessaria nessuna generazione di essi. Si passerà direttamente all'operazione per l'esecuzione dei test (passo 10.);
6. *Verifica della tipologia di test della pagina*: nel caso di test del sito differenti dal funzionale, è necessario verificare se il test della pagina sarà l'apertura di un collegamento oppure la compilazione di un modulo (configurazione $TipoTest = "FORMPAGE"$) . Nel primo caso si potrà proseguire per la creazione dei test (passo 10.), nel secondo sarà necessario effettuare l'analisi del modulo web;
7. *Analisi della pagina*: nel caso il test sia la compilazione di un modulo, in questa operazione lo strumento si posiziona sulla pagina da testare dopo aver eseguito le eventuali fasi preliminari configurate. Raggiunta la pagina, se è stato configurato un identificativo del form, obiettivo del test attraverso la configurazione $ContainerElementActio$, verrà ricercato, altrimenti si analizzerà il primo modulo

trovato nella pagina. Per analisi si intende la raccolta di tutti gli elementi HTML di tipo input per permettere la compilazione del form. Per questa operazione verranno anche considerate le configurazioni $ConsideraCampiSolaLettura$ e $ConsideraCampiNascosti$ per definire se effettuare i test anche su queste tipologie di campi oppure no. Finita l'analisi si svolgono le eventuali fasi finali configurate;

8. *Generazione dei valori considerando gli elementi trovati e quelli configurati*: considerando gli elementi trovati, gli elementi configurati attraverso il file di configurazione oppure attraverso il file dei dati, verranno generati i valori. Per la generazione si considera il tipo di test del sito impostato e si analizzerà il dominio per ogni singolo elemento. Tale dominio se non è definito quello funzionale si considera quello strutturale per la tipologia di campo HTML riscontrata. È anche possibile nel file di configurazione aver definito il metodo di generazione del valore con i rispettivi parametri da utilizzare;
9. *Creazione delle combinazioni*: in questa operazione si è adottato il criterio $All\ Combination$ (letteralmente "tutte le combinazioni"), menzionato nel paragrafo 2.4.2.1 - Black-Box Testing, per combinare i valori calcolati per ogni caratteristica del dato di input. È stato scelto per sfruttare l'automatismo in quanto il criterio è oneroso da un punto di vista computazionale ed inoltre poiché gli altri criteri possono essere impostati attraverso la definizione di un test del sito funzionale ed il filtraggio dei test impostando opportunamente il file dei dati;
10. *Creazione dei test*: in questa fase si crea la struttura per l'esecuzione dei test, impostando eventuali pagine da raggiungere o moduli da compilare con i relativi valori da utilizzare;
11. *Esecuzione dei test*: per ogni test creato, vengono eseguite le fasi iniziali, viene eseguito il test, viene controllato il nuovo contenuto della pagina cercando il testo nella configurazione $RisultAspOK$. In caso non venga trovato, si verifica che non sia presente neanche il testo della configurazione $RisultAspOK$ discriminando l'esito positivo o negativo del test. Concludendo con le eventuali fasi finali, successivamente vengono memorizzati nel file dei dati i valori di input mentre il risultato finale del test verrà memorizzato nel file di report. Eventuali errori applicativi faranno fallire il test e verranno riportati nel report e nel file di log.



Algoritmo 4.2 - Flusso di operazioni svolte dallo strumento Automatic Tester of Web Pages

4.3.5. Note di sviluppo ed utilizzo

Lo strumento è stato sviluppato in C# framework 4.0, la soluzione è stata divisa in due progetti, l'eseguibile come *Console Windows* ed una libreria, il motore dello strumento.

Esso è stato pensato per essere il più versatile ed espandibile possibile.

Si è sfruttata la tecnica "Custom Section Configuration" [42] perché oltre ad avere il vantaggio di poter rappresentare una configurazione complessa in un unico file XML, la sua struttura ad interfacce permetterà in futuro di poter adottare tecniche di sviluppo nel creare oggetti di configurazioni ed utilizzare lo strumento dall'esterno e non soltanto dall'eseguibile.

Per avere un riscontro dell'esecuzione dei test in tempo reale, è stata utilizzata la nota libreria *log4net* [43], ed è possibile modificare le proprietà del file di log, risultato di ciò che si vuole tracciare, modificando il file di configurazione *log4net.config* posizionato nella stessa cartella dell'eseguibile del programma.

La classe contenente le principali funzioni per la generazione degli input è stata creata come "*static partial class*"

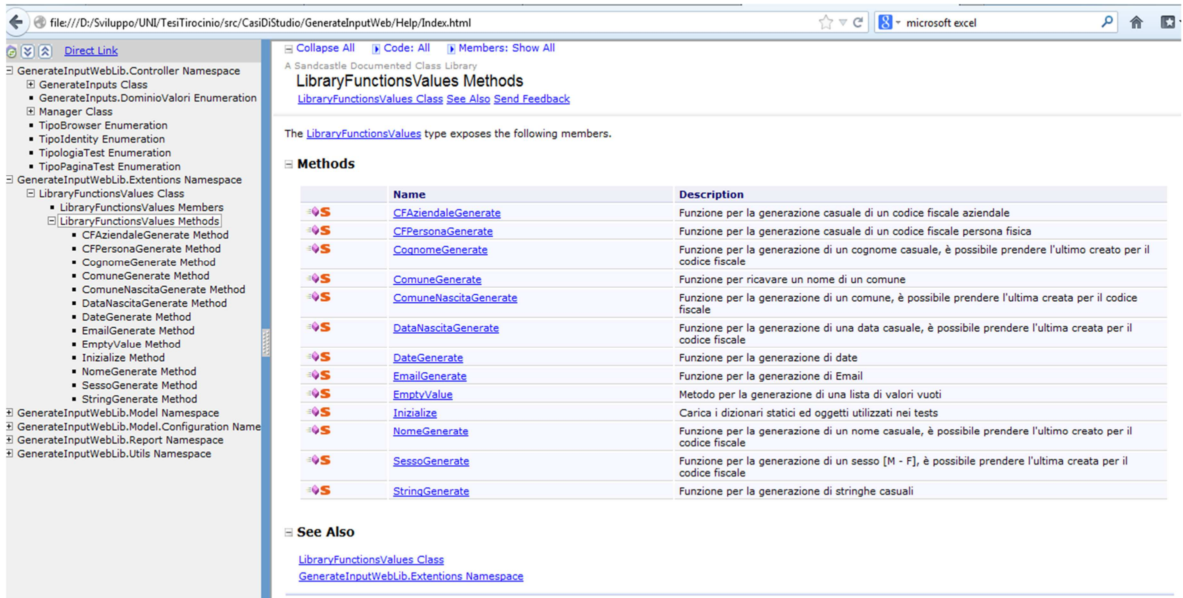
public static partial class LibraryFunctionsValues

per dare la possibilità a futuri sviluppatori di aggiungere ulteriori classi parziali arricchite di funzioni di generazione degli input personalizzate con il solo vincolo che i metodi rispecchino la seguente firma

public static string[] Metodo(string[] parameters)

in modo che sia richiamabile dal file di configurazione.

Sfruttando il linguaggio C# per la creazione della documentazione ed un applicativo sviluppato da Microsoft, SandCastle [44] , si è creato un sito web interno al progetto per poter consultare meglio la documentazione e le API sviluppate, in questo modo si potrà sapere, per esempio, quali metodi sono già stati sviluppati per la generazione degli input.



Esempio 4.4 - Sito web per la consultazione delle API dello strumento

Al momento l'unico avvio possibile è quello a linea di comando.

$$GenerateInputWeb\backslash GenerateInputWeb.Execute\backslash bin\backslash Release \\ \backslash GenerateInputWeb.Execute.exe\ FileDiConfigurazione.config$$

Esempio 4.5 - Avvio di Automatic Tester of Web Pages a linea di comando

Un esempio di utilizzo è il seguente:

1. Creare un semplice file di configurazione per effettuare un test di tipo strutturale ed eseguire il test;
2. Articolare il file di configurazione precedente, secondo i risultati trovati;
3. Utilizzando il file di dati generato dalle esecuzioni precedenti ed i nomi dei campi trovati, effettuare i test funzionali.

4.3.6. Sviluppi da ultimare

Considerando gli obbiettivi preposti, descritti nel paragrafo 4.3.1 - Introduzione, si sono raggiunti quasi tutti gli obbiettivi tranne i seguenti:

- *Possibilità di avere un Assert in funzione del valore del campo*: al momento si verifica la presenza di un testo nella pagina raggiunta dal test, in futuro si vorrebbe avere una funzione legata ai valori;
- *Partendo da una pagina, visitare tutte le pagine collegate*: l'utilizzo dello strumento per puri test strutturali, come un *crawler* [43] per verificare la raggiungibilità di tutte le pagine;
- *Generazione di input per pagine multi-form*: al momento per le pagine formate da più moduli in sequenza è solo possibile effettuare test di tipo funzionali, impostando i valori da utilizzare nei singoli moduli;
- *Finire l'implementazione di generazione dei valori dentro e fuori i domini funzionali*: sono ancora da completare i metodi di generazione per tutti i domini definiti, sono stati sviluppati quelli utilizzati nei casi di studio;
- *Integrare la documentazione*: il codice è stato largamente commentato e documentato ma sarebbe necessario un ampliamento della documentazione per aiutare nella creazione del file di configurazione per specifici test.

[43] Crawler : Anche detto *spider* (letteralmente ragno) è un tipo di applicazione che come un ragno visita una ragnatela, in ambito informatico l'applicativo visita tutti i collegamenti di una applicazione web, potrebbe andare anche all'infinito visitando tutta la rete Internet se partisse da un sito web collegato a centinaia di altri siti.

CAPITOLO 5 Analisi sperimentale dei casi di studio

In questo capitolo verrà evidenziato l’aspetto pratico degli studi effettuati. I vari strumenti analizzati sono stati comparati ed utilizzati in ambito lavorativo presso l’azienda ETT s.r.l..

5.1. White-box Testing

Le verifiche strutturali trattate nel paragrafo 2.4.2.2 - White-Box Testing, hanno evidenziato l’importanza di avere un codice stabile e ben sviluppato portando un vantaggio sul progetto fin dall’avvio dello sviluppo.

In questo capitolo vengono riportate le prove sperimentali effettuate ed i risultati riscontrati con lo strumento Pex.

5.1.1. Microsoft Pex a confronto con i software di ricerca sui sorgenti Benchmarks Hand-Crafted

Per poter valutare la bontà dello strumento Pex, si è deciso di confrontare la sua esplorazione con quelle eseguite dagli strumenti sperimentali descritti nel precedente paragrafo 3.1.2.1 - Software sperimentali sviluppati analizzando un insieme di sorgenti. La valutazione si è basata sul numero di test generati e sulla copertura raggiunta.

5.1.1.1. Sorgenti analizzati

Sono stati analizzati un insieme di sorgenti "*Benchmarks Hand-Crafted*" (letteralmente “sorgenti di riferimento fatti a mano") sviluppati in linguaggio C. Poiché Pex supporta solo il C#, è stato fatto il porting[44] in C# controllando che il grafo del flusso di controllo non variasse. Altra caratteristica importante è la mancata dipendenza da ambienti esterni o

[44] Porting: Letteralmente portabilità, nell’informatica è la procedura di adattamento di un componente software da un sistema ad un altro. In questo caso è una modifica del linguaggio utilizzato per l’esecuzione di un algoritmo.

punti irraggiungibili. Essi comprendono un insieme di metodi creati in una unica classe in modo automatico. Ogni metodo è stato creato con il seguente algoritmo

```
BENCH(int i_1, int i_2, ..., int i_n)
    i_n - k + 1 = ( MEANVALUE(i_1, i_2, ..., i_n - k) + R_1 ) MODULE (m+1)
    i_n - k + 2 = ( MEANVALUE(i_1, i_2, ..., i_n - k) + R_2 ) MODULE (m+1)
    i_n = ( MEANVALUE(i_1, i_2, ..., i_n - k) + R_k ) MODULE (m+1)
    if (i_1 == 0)
        body_{1,1}
    else if (i_1 == 1)
        body_{1,2}
    ...
    else if (i_1 == (m-1))
        body_{1,m-1}
    else
        body_{1,m}
    ...
    if (i_n == 0)
        body_{n,1}
    ...
    else
        body_{n,m}
    return
```

Algoritmo 5.1 - Creazione dei metodi da testare

dove ogni metodo è caratterizzato da tre parametri:

n: è il numero di variabili in ingresso del metodo

m: è il numero di blocchi decisionali in cascata per variabile

k: è il numero di variabili all'interno del metodo che dipendono dalle variabili in ingresso

Per ogni metodo così generato ne sono stati creati dieci mescolando l'ordine dei blocchi in modo casuale.

Per esempio, impostando i parametri

$$n: 5, m: 2, k: 1, rnd: 0\ (\,primo\ casuale\,)$$

si è creato il metodo

```
public void synth_m2_n5_k1_rnd0(int par0, int par1, int par2, int par3)
{
  int par4 = ((par0 + par1 + par2 + par3) / 4 + 0) % 3;
  if (par2 == 1) { ; }
  else if (par2 == 0) { ; }
  else { ; }
  if (par3 == 1) { ; }
  else if (par3 == 0) { ; }
  else { ; }
  if (par4 == 0) { ; }
  else if (par4 == 1) { ; }
  else { ; }
  if (par1 == 0) { ; }
  else if (par1 == 1) { ; }
  else { ; }
  if (par0 == 1) { ; }
  else if (par0 == 0) { ; }
  else { ; }
}
```

Esempio 5.1 - Metodo testato

I metodi totali creati ed analizzati sono stati 2200.

5.1.1.2. Risultati e valutazioni di Pex

Pex è stato avviato nell'analisi dei sorgenti "*Benchmarks Hand-Crafted*" utilizzando strategie, opzioni e modalità di avvio differenti.

5.1.1.2.1. Descrizione delle esplorazioni effettuate

Di seguito la descrizione degli avvii effettuati. Ad ogni esplorazione è stato dato un nome come identificativo. Le strategie utilizzate sono state illustrate nel paragrafo 3.2.2.3 - Strategie.

- $Pex - Default - VS2010$: avvio di Pex da Microsoft Visual Studio 2010 utilizzando la strategia di $Default$;
- $Pex - Default2$: avvio di Pex da linea di comando utilizzando la strategia di $Default$;
- $Pex - Default3$: avvio di Pex da linea di comando utilizzando la strategia di $Default$, per una seconda volta;
- $Pex - Default4_MaxSolver10$: avvio di Pex da linea di comando utilizzando la strategia di $Default$ con l'opzione del tempo di risoluzione del solver a dieci secondi anziché uno;
- $Pex - Default5 - VS2010$: avvio di Pex da Microsoft Visual Studio 2010 utilizzando la strategia di $Default$, per la seconda volta;
- $Pex - FrontierDepthFirst$: avvio di Pex da linea di comando utilizzando la strategia "$Frontier\ Depth\ First$";
- $Pex - FrontierBreadthFirst$: avvio di Pex da linea di comando utilizzando la strategia "$Frontier\ Breadth\ First$";
- $Pex - FrontierIterativeDeepening$: avvio di Pex da linea di comando utilizzando la strategia "$Frontier\ Iterative\ Deepening$";
- $Pex - FrontierRandom$: avvio di Pex da linea di comando utilizzando la strategia "$Frontier\ Random$".

Le esplorazioni hanno avuto una durata media di due giorni ed è stata utilizzata una macchina virtuale dedicata con CPU Dual-Core da 2.53 GHz con 2G RAM[45].

5.1.1.2.2. Risultati delle esplorazioni di Pex messe a confronto con quelle degli strumenti sperimentali analizzati

[45] CPU Dual-Core, RAM : Componenti elettronici per descrivere la potenza di calcolo e di memoria di un computer

Mediamente Pex non si è comportato in modo ottimale, è molto sensibile alle risorse a sua disposizione. Solo la strategia di $Default$ si è rilevata utile mentre le altre hanno mediamente raggiunto coperture parziali.

Durante l'esplorazione $Pex - Default - VS2010$ la macchina virtuale ha avuto un calo di prestazioni, anche Visual Studio è andato in errore e l'esplorazione è stata considerata inizialmente poco attendibile. Quella denominata $Pex - Default2$ ha raggiunto la percentuale di copertura massima per più volte. Anche durante l'esplorazione $Pex - Default3$ la macchina virtuale ha avuto un calo di prestazioni e nel file di log sono stati registrati i seguenti messaggi significativi sul raggiungimento della memoria massima a disposizione:

$$< boundary > \ maxworkingset \ - \ 555MB \ (exceeded\ allowed\ working\ set)$$

$$< boundary > \ maxworkingset \ - \ 555MB \ (exceeded\ allowed\ working\ set)$$

$$< boundary > \ maxworkingset \ - \ 552MB \ (exceeded\ allowed\ working\ set)$$

$$< boundary > \ maxworkingset \ - \ 539MB \ (exceeded\ allowed\ working\ set)$$

Poiché nelle prime tre esecuzioni erano stati riscontrati diversi messaggi inerenti al tempo scaduto del solver per la risoluzione dei vincoli

$$< boundary > \ MaxConstraintSolverTime, 28\ times$$

l'esplorazione $Pex - Default4_MaxSolver10$ è stata predisposta per utilizzare un tempo maggiore, pensando che avrebbe migliorato l'esplorazione rispetto alle precedenti, invece non è stata quella ottimale. Infine l'esplorazione $Pex - Default5 - VS2010$, replica della prima $Pex - Default5 - VS2010$, si è comportata peggio di essa nonostante Visual Studio non sia andato in errore, ma comunque mediamente meglio di altre.

Di seguito una tabella con le percentuali raggiunte.

Esplorazione	**Copertura**
Pex-Default2	99,19%
Pex-Default-VS2010	99,05%
Pex-Default5-VS2010	98,79%
Pex-Default4_MaxSolver10	97,88%
Pex-FrontierBreadthFirst	93,18%
Pex-Default3	84,95%

Pex-FrontierRandom	84,36%
Pex-FrontierDepthFirst	60,10%
Pex-FrontierIterativeDeepening	32,07%

Tabella 5.1 - Risultati delle esplorazioni di Pex sui sorgenti "Benchmarks Hand-Crafted"

Considerando le migliori tre esplorazioni, esse sono state confrontate con i risultati sperimentali riscontrati con gli strumenti da laboratorio.

Dei 2200 metodi si è focalizzata l'attenzione solo su un sottoinsieme di essi, di cui si riportano i risultati nella tabella sottostante.

Complessità			TeGeVe	Sage	SAT&PREF	noPref	Pex-Default-VS2010		Pex-Default2		Pex-Default5-VS2010	
m	n	k	T. Tot	T. Tot	T. Tot	T. Tot	T. Tot	Cov.	T. Tot	Cov.	T. Tot	Cov.
2	5	0	3	12	3	11	11,00	100,00%	11,00	100,00%	11,00	100,00%
2	5	1	3	11	4	10	9,64	100,00%	9,18	100,00%	9,64	100,00%
2	5	2	3	9	3	9	8,45	100,00%	8,45	100,00%	8,45	100,00%
2	5	3	3	9	4	7	6,82	100,00%	6,91	100,00%	6,82	100,00%
2	5	4	3	6	4	5	4,91	100,00%	4,91	100,00%	4,91	100,00%
2	10	0	3	35	3	19	21,00	100,00%	21,00	100,00%	21,00	100,00%
2	10	1	3	21	3	18	17,82	100,00%	18,00	100,00%	17,82	100,00%
2	10	2	3	17	3	17	15,64	100,00%	15,18	100,00%	15,64	100,00%
2	10	3	3	21	3	15	13,55	100,00%	13,45	100,00%	13,55	100,00%
2	10	4	3	14	3	14	11,45	100,00%	11,82	100,00%	11,45	100,00%
2	10	5	3	13	4	12	10,55	100,00%	10,45	100,00%	10,55	100,00%
2	10	6	3	12	3	10	9,00	100,00%	8,73	100,00%	9,00	100,00%
2	10	7	3	9	4	8	8,18	100,00%	7,82	100,00%	8,18	100,00%
2	10	8	3	8	4	6	6,91	100,00%	6,64	100,00%	6,91	100,00%
2	10	9	3	6	3	5	4,82	100,00%	4,64	100,00%	4,82	100,00%
4	5	0	5	21	5	21	21,00	100,00%	21,00	100,00%	21,00	100,00%
4	5	1	7	22	6	20	19,64	100,00%	19,45	100,00%	19,64	100,00%
4	5	2	6	25	7	16	15,09	100,00%	15,09	100,00%	15,09	100,00%
4	5	3	7	20	6	13	12,27	100,00%	11,73	100,00%	12,27	100,00%
4	5	4	7	11	6	7	8,09	100,00%	8,00	100,00%	8,09	100,00%
4	10	0	5	69	5	30	41,00	100,00%	41,00	100,00%	41,00	100,00%
4	10	1	8	60	5	34	37,45	100,00%	38,00	100,00%	37,45	100,00%
4	10	2	7	110	5	32	31,45	100,00%	31,18	100,00%	31,45	100,00%
4	10	3	8	72	5	30	27,00	100,00%	27,91	100,00%	27,00	100,00%
4	10	4	7	52	6	26	24,73	100,00%	24,36	100,00%	24,73	100,00%
4	10	5	7	57	6	24	21,18	100,00%	21,64	100,00%	21,18	100,00%
4	10	6	7	57	6	20	18,36	100,00%	18,27	100,00%	18,36	100,00%
4	10	7	7	33	6	16	14,45	100,00%	14,82	100,00%	14,45	100,00%
4	10	8	7	14	7	12	12,09	100,00%	11,64	100,00%	12,09	100,00%
4	10	9	7	10	6	7	8,18	100,00%	8,00	100,00%	8,18	100,00%
6	5	0	7	31	7	31	31,00	100,00%	31,00	100,00%	31,00	100,00%
6	5	1	11	43	9	29	27,45	100,00%	28,09	100,00%	27,45	100,00%
6	5	2	11	38	9	23	22,73	100,00%	22,45	100,00%	22,73	100,00%
6	5	3	10	34	9	18	17,91	100,00%	17,45	100,00%	17,91	100,00%
6	5	4	10	16	8	10	10,91	100,00%	10,45	100,00%	10,91	100,00%
6	10	0	7	88	7	60	61,00	100,00%	61,00	100,00%	61,00	100,00%
6	10	1	11	155	8	53	52,82	100,00%	53,09	100,00%	52,82	100,00%
6	10	2	11	196	8	51	47,09	99,77%	46,73	100,00%	44,73	96,62%
6	10	3	10	230	8	43	43,09	100,00%	44,00	100,00%	43,09	100,00%
6	10	4	11	123	8	39	37,55	100,00%	37,27	100,00%	37,55	100,00%
6	10	5	11	146	8	32	32,55	100,00%	32,82	100,00%	32,55	100,00%
6	10	6	10	73	9	30	27,73	100,00%	27,55	100,00%	27,82	100,00%
6	10	7	11	60	9	23	22,36	100,00%	22,18	100,00%	22,36	100,00%
6	10	8	11	31	9	18	17,09	100,00%	17,27	100,00%	17,09	100,00%
6	10	9	11	15	8	11	11,91	100,00%	11,64	100,00%	11,91	100,00%
Medie			**6,67**	**47,00**	**5,82**	**21,00**	**20,78**	**99,99%**	**20,74**	**100,00%**	**20,73**	**99,92%**
Totali Test			**300**	**2115**	**262**	**945**	**934,91**		**933,27**		**932,64**	

Tabella 5.2 - Tabella comparativa di Pex con gli strumenti sperimentali

Poiché per ogni tipologia di metodo, rappresentato dai valori dei parametri s, k e b, Pex si è comportato in modo diverso per ogni singolo metodo, si è fatta la media aritmetica del numero dei test generati e della copertura raggiunta. L'esplorazione $Pex - Default2$ è stata l'unica a raggiungere la copertura massima del 100% per questo gruppo di metodi

analizzati. Per gli strumenti sperimentali non è stata riportata la percentuale di copertura in quanto è sempre stata del 100%. Non è stato riportato il tempo di esecuzione in quanto le esecuzioni degli strumenti universitari sono state effettuate in macchine completamente diverse rispetto a quella dedicata per le esecuzioni di Pex. Pex rispetto a SAGE (l'unico strumento ad utilizzare DSE) si è comportato in modo più efficiente in quanto sono stati generati meno test per raggiungere la copertura massima. La differenza tra SAGE è Pex, è che SAGE è pensato per testare “unmanaged code” (letteralmente “codice non gestito”) considerando l’applicativo in forma binaria, mentre Pex analizza “managed code” (letteralmente “codice gestito” come evidenziato nel paragrafo 3.2.2.2.1 - Instrumentation considerando codice interpretato da MSIL .NET prima della compilazione.

Rispetto agli altri strumenti Pex ha generato un numero maggiore di test in quanto non esiste un concetto di preferenza nonostante vi sia una funzione di guadagno che distribuisce le priorità sui percorsi già visitati precedentemente. Per esempio eseguendo l'analisi del seguente metodo :

```
public void Test(bool a, bool b)
{
 if (a)
 {
  a = True;
 }
 else
 {
  a = False;
 }
 if (b)
 {
  b = True;
 }
 else
```

$\{$

$b = False;$

$\}$

$\}$

Pex genererà i seguenti test:

$Test(FALSO, FALSO)$

$Test(FALSO, VERO)$

$Test(VERO, FALSO)$

mentre basterebbero solo questi :

$Test(FALSO, VERO)$

$Test(VERO, FALSO)$

Come era prevedibile lo strumento SAT&PREF si è rilevato il migliore per questo tipo di analisi.

Strumento	N. Test Totali Generati
SAT&PREF	262
TeGeVe	300
Pex-Default2	933,27
noPref	945
SAGE	2115

Tabella 5.3 – Classifica degli strumenti sperimentali con Pex nell'analisi della porzione scelta dei sorgenti "Benchmarks Hand-Crafted"

5.1.2. Microsoft Pex sul caso di studio di ETT s.r.l.

Nel lavoro svolto presso ETT s.r.l. lo strumento Pex è stato utilizzato per l'analisi di due progetti. Di seguito una breve spiegazione di come è stato applicato e quali benefici ha portato.

5.1.2.1. Applicazione Archivia Log

Nel tempo molte applicazioni web sviluppate dall'azienda creavano numerosi "file di log" dovuti alle numerose richieste ricevute e dalla verbosità alta delle loro configurazioni. Per

poter archiviare tali log è stata creata una semplice applicazione che dopo aver configurato una cartella contenente i file da spostare, una espressione regolare per filtrare il nome dei file e la cartella di destinazione, al suo avvio identifica i file, crea un file compresso contenente i file trovati e lo sposta nella cartella di destinazione.

5.1.2.1.1. Spiegazione dell'utilizzo di Pex sul caso di studio Archivia Log

L'applicazione è composta da un programma di avvio e da una libreria con soli due metodi. Il principale metodo

viene richiamato dall'applicativo per la definizione di cosa si vuole fare.

```
//Esecuzione
public static int eseguiSpostamento(string dirLog,     string dirBackup,
            string patternFindLog,     bool unicoFile)
```

Un secondo metodo, richiamato dal primo, crea il file compresso.

```
// dato il fileName da zippare ritorna il nuovo nome file
private static void zipFile(string[] fileNames, string dirBackup)
```

Inizialmente il metodo "*zipFile*" era pubblico, all'avvio di Pex esso veniva visitato due volte, essendo richiamato dal metodo "*eseguiSpostamento*" . Si è preferito renderlo privato in quanto utilizzato solo in questo contesto.

Al primo avvio è stato subito riscontrato un problema, la possibilità di passare al metodo "*eseguiSpostamento*" parametri vuoti.

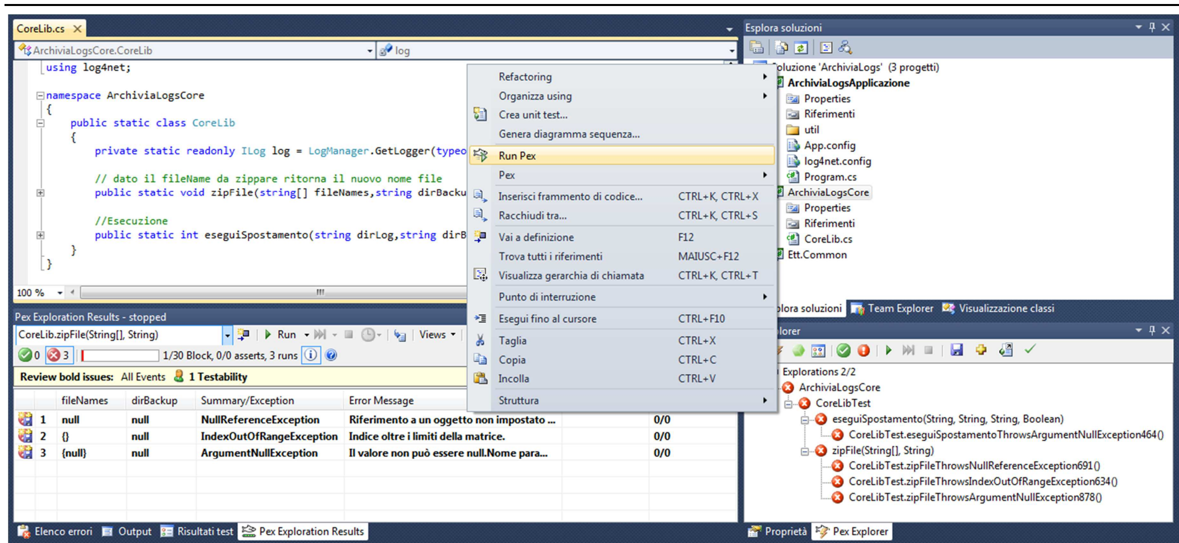


Figura 5.1 - Avvio di Pex

Nel caso il parametro con stringa vuota rappresentava la cartella di destinazione, si rischiava di creare il file compresso in una cartella sbagliata oppure se la stringa vuota era la cartella dei file di log si rischiava cercare i file di log in una cartella diversa da quella corretta, spostando file sbagliati.

Seguendo il suggerimento di Pex " *Add Precondition* " (letteralmente “aggiungi precondizione”) e modificando l'eccezione con un messaggio più specifico di quello proposto, all'avvio successivo Pex ha proseguito nella sua esplorazione.

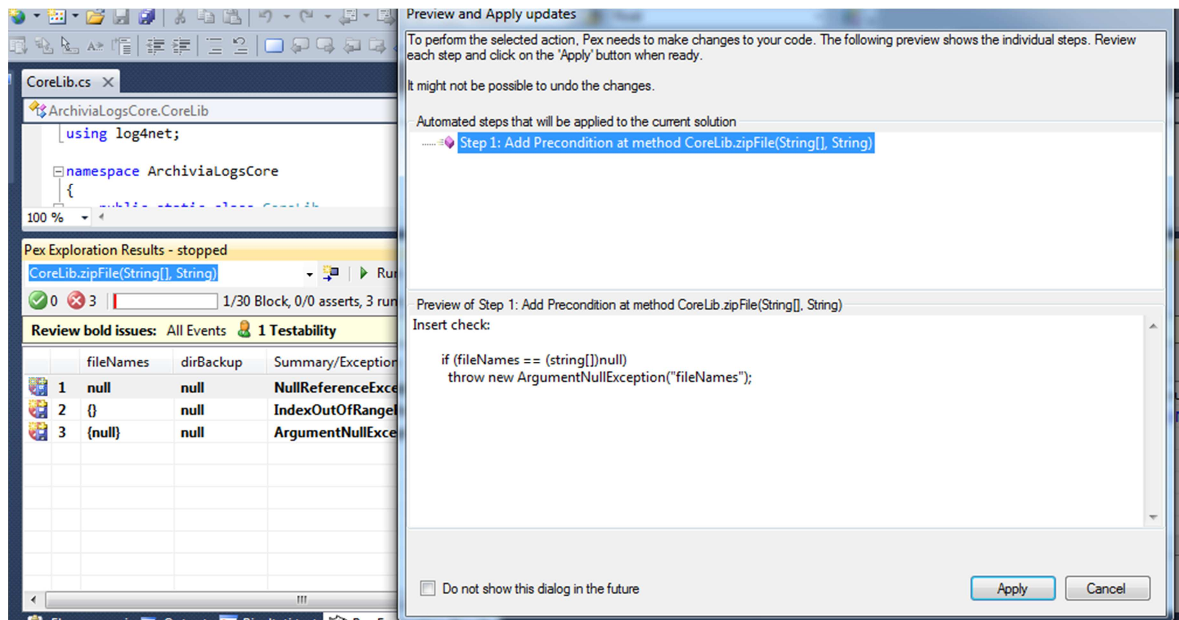


Figura 5.2 - Aggiungi precondizione

Analizzando il costruttore "$System.IO.FileInfo$", facendo parte del framework C# o comunque essendo un oggetto al di fuori del progetto, lo strumento segnala che il costruttore non è testabile.

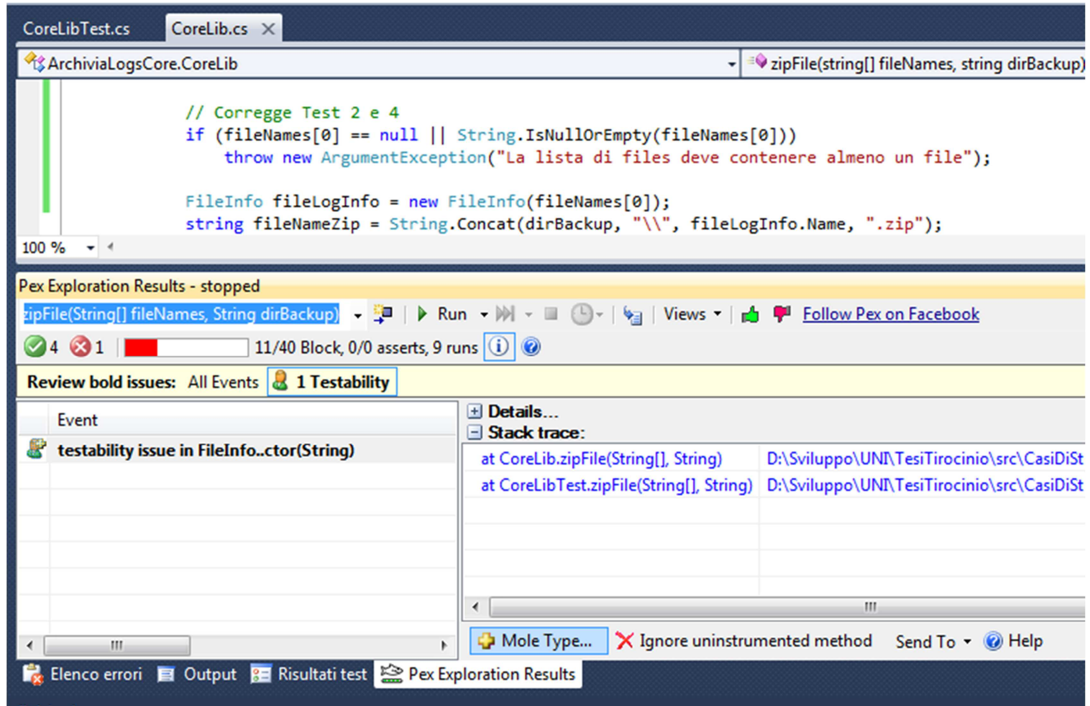


Figura 5.3 - Metodo non testabile

Scegliendo il suggerimento di applicare l'isolamento "$Mole\ Type$" attraverso Moles [21],

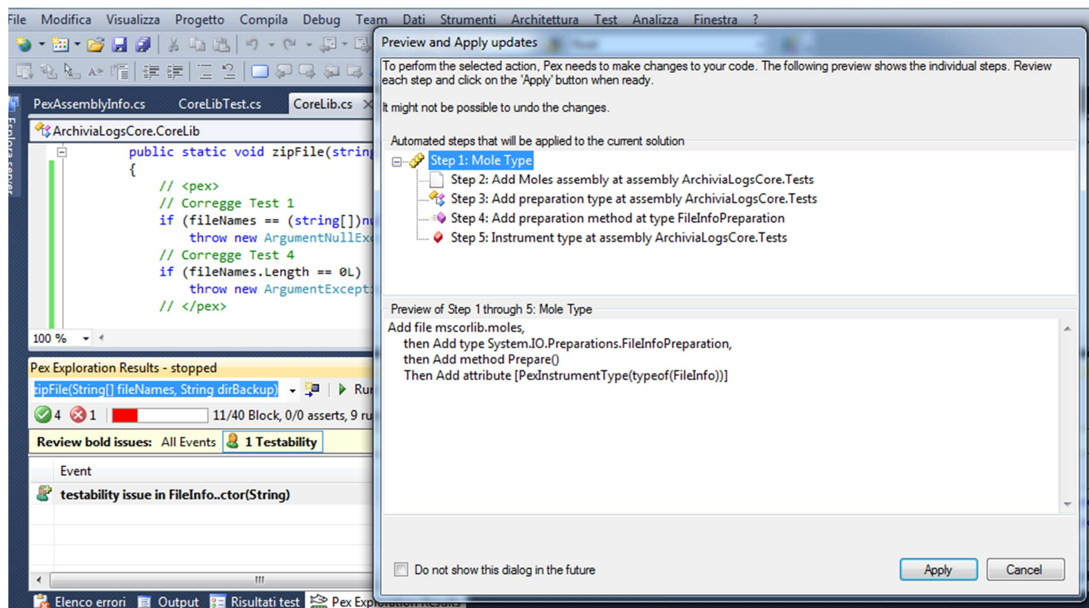


Figura 5.4 - Isolamento metodo non testabile

viene predisposta una nuova classe chiamata *System.IO.Preparations.FileInfoPreparation* con il metodo "*prepare*". È stato ridefinito il costruttore *FileInfo* e Pex ha proseguito nell'esplorazione.

È risuccessa la stessa cosa per le classi *System.IO.DirectoryInfo* e *System.IO.Directory*.

Durante la prima esecuzione è anche stata salvata la suite di test, utile per i successivi test per la verifica nella non regressione del codice e per poter salvare ed applicare i suggerimenti di Pex.

Di seguito i listati delle classi suggerite da Pex e sviluppate secondo le esigenze necessarie per proseguire nei test.

FileInfoPreparation.cs

```
// < copyright file = "FileInfoPreparation.cs" company = "Microsoft"
            > Copyright © Microsoft 2012 </copyright >


using Microsoft.Pex.Framework;
using System.IO.Moles;


namespace System.IO.Preparations
{
  /// < summary > Contains a method to prepare the type FileInfo
            </summary >
  public static partial class FileInfoPreparation
  {
   /// < summary >
> Prepares the environment (and the moles)before executing any method of
the prepared type FileInfo </summary >
    [PexPreparationMethod(typeof(FileInfo))]
    public static void Prepare()
    {
```

```
            string fileName = "AdapterWS_Errors.2012-01-01";

            string dirFileName
                = @"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
                \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
                \Tests";


            MFileInfo.BehaveAsCurrent();

            MFileInfo.ConstructorString = (@this, a) =>

            {

            };


            MFileInfo.AllInstances.NameGet = (@this) =>

            {

              return fileName;

            };


            MFileInfo.AllInstances.DirectoryNameGet = (@this) =>

            {

              return dirFileName;

            };

          }

        }

      }
```

____

DirectoryPreparation.cs

```
      using Microsoft.Pex.Framework;

      using System.IO.Moles;


      namespace System.IO.Preparations
```

```
{
  /// < summary > Contains a method to prepare the type Directory
             </summary >
  public static partial class DirectoryPreparation
  {
 /// < summary >
> Prepares the environment (and the moles)before executing any method of
 the prepared type Directory </summary >
    [PexPreparationMethod(typeof(Directory))]
    public static void Prepare()
    {
      MDirectory.BehaveAsCurrent();
      MDirectory.GetFilesStringString = (a, b) =>
      {
        /// Errore1uscita dal comando
        /// ""C:\Program Files\Microsoft Moles\bin\moles.exe" @"D:
             \Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
             \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
             \obj\Release\Moles\moles.args"" con codice
             − 1002. ArchiviaLogsCore.Tests
        //
/ return new string[] { String.Concat(dirFileName, "TextFile1.txt"),
String.Concat(dirFileName, "TextFile2.txt") };
        return new string[] {
          @"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
             \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
             \Tests\AdapterWS_Errors.2012 − 01 − 01",
          @"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
             \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
             \Tests\TextFile1.txt",
          @"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio
             \Elementi COAP 2.3.1\ArchiviaLogs\ArchiviaLogsCore.Tests
             \Tests\AdapterWS_Errors.2012 − 01 − 03"
```

```
            };
        };
        MDirectory.ExistsString = (a) =>
        {
            return false;
        };
        MDirectory.SetCurrentDirectoryString = (a) =>
        {

        };
        MDirectory.CreateDirectoryString = (a) =>
        {
            return new DirectoryInfo(a);
        };
      }
    }
  }
```

DirectoryInfoPreparation.cs

```
using Microsoft.Pex.Framework;
using System.IO.Moles;

namespace System.IO.Preparations
{
  /// <summary> Contains a method to prepare the type DirectoryInfo
            </summary>
  public static partial class DirectoryInfoPreparation
  {
```

```
    /// < summary >
  > Prepares the environment (and the moles)before executing any method of
  the prepared type DirectoryInfo </summary >
    [PexPreparationMethod(typeof(DirectoryInfo))]
    public static void Prepare()
    {
      MDirectoryInfo.BehaveAsCurrent();
      MDirectoryInfo.ConstructorString = (@this, a) =>
      {


      };
    }
  }
}
```

5.1.2.1.2. Risultati riscontrati delle esplorazioni sul caso di studio Archivia Log

Dopo varie iterazioni, l'applicazione dei suggerimenti ad ogni esplorazione e conseguenti sviluppi, si è arrivati all'ultima esplorazione senza suggerimenti. Le variazioni sono state minime, hanno comportato l'aggiunta del controllo di non poter richiamare il metodo "*eseguiSpostamento*" ed il cambio di visibilità da pubblico a privato del metodo "*fileZip*".

Prima Dopo

Figura 5.5 - Differenze della libreria utilizzata in “Archivia Log” prima e dopo l’utilizzo di Pex

È stata raggiunta una copertura del codice del 89.02% , l'applicazione ha molte dipendenze con fattori esterni pertanto gli oggetti esterni per cui è stato creato un "oggetto fittizio", non hanno potuto coprire tutte le casistiche.

```
public static int eseguiSpostamento(string dirLog,string dirBackup,string patternFindLog,bool unicoFile)
prev|top{
    // trova file
    //NOT WORK: string[] listFile = Directory.GetFiles(dirLog, "server.log.[0-9]{4}-[0-9]{2}-[0-9]{2}");

    if (String.IsNullOrEmpty(dirLog) || String.IsNullOrEmpty(dirBackup) || String.IsNullOrEmpty(patternFindLog))
        throw new ArgumentNullException(
            String.Format("Attenzione uno dei seguenti parametri è nullo. dirLog={0},dirBackup={1},patternFindLog={2}",dirLog,dirBackup,patternFindLog));

    string[] listFile = Directory.GetFiles(dirLog, patternFindLog);
    if(listFile.Length > 0)
    {
        // Ricava data per archivio
        DateTime fileDateFirst = DateTime.Now;
        string fileNameFirst = listFile[0];
        try
        {
            fileDateFirst = DateTime.Parse(fileNameFirst.Substring(fileNameFirst.Length - 10, 10));
        }
        catch (Exception e)
        {
            // use Now if no date
        }
        var dirBackupFile = String.Concat(dirBackup, "\\", fileDateFirst.Year, "\\", fileDateFirst.Year, "_",
                fileDateFirst.Month <= 9 ? (object)String.Concat("0", fileDateFirst.Month) : fileDateFirst.Month);
        if (!Directory.Exists(dirBackupFile))
        {
            Directory.CreateDirectory(dirBackupFile);
        }
        if (unicoFile)
        {
            zipFile(listFile, dirBackupFile);
        }
        else
        {
            foreach (var fileName in listFile)
            {
                DateTime fileDate = DateTime.Now;
                try
                {
                    fileDate = DateTime.Parse(fileName.Substring(fileName.Length - 10, 10));
                }
                catch (Exception e)
                {

                }
                dirBackupFile = String.Concat(dirBackup, "\\", fileDate.Year, "\\", fileDate.Year, "_",
                    fileDateFirst.Month <= 9 ? (object)String.Concat("0", fileDate.Month) : fileDate.Month);
                if (!Directory.Exists(dirBackupFile))
                {
                    Directory.CreateDirectory(dirBackupFile);
                }
                string[] file = new string[1];
                file[0] = fileName;
                zipFile(file, dirBackupFile);
            }
        }
    }

    return listFile.Length;
}
}
```

Figura 5.6 - Copertura del codice

Infine con cinque test, il report finale riporta gli avvii e dettagli dei parametri utilizzati evidenziando la mancanza di errori (Figura 3.6 - Sito web del report prodotto dall'esplorazione) .

5.1.2.2. Libreria Ett.Common

Dopo l'esperienza dell'utilizzo di Pex con un progetto semplice come l'applicazione "Archivia Logs", si è deciso di applicare Pex ad una libreria sviluppata negli anni all'interno di ETT ed utilizzata in diversi progetti, tra cui l'applicazione precedentemente testata. Anche questa libreria dipende molto da ambienti esterni ed è formata da un insieme di classi per facilitare l'accesso ai database, la lettura di file, l'implementazione di funzioni comuni di sicurezza e la scrittura di file di log.

5.1.2.2.1. Utilizzo di Microsoft Pex sul caso di studio della libreria Ett.Common

La libreria analizzata è molto complessa, si è deciso di non avviare Pex su tutto il progetto ma per namespace differenti.

Aprendo con Visual Studio il progetto dei sorgenti ed aprendo il sorgente di una classe, con il cursore posizionato tra il namespace dichiarato ed il nome della classe, cliccando il tasto destro del mouse nel menù presentato premendo su "*Run Pex*", l'esplorazione viene avviata su tutte le classi appartenenti al namespace della classe visitata.

Le iterazioni sono state molto numerose. Di seguito alcuni casi particolari.

Nell'esplorazione del namespace *Ett.Common.DataBase* per la classe *DataBaseManager*, utilizzata come interfacciamento per l'accesso ai database, è emerso che un suo metodo era richiamato in modo ricorsivo su se stesso, grazie a Pex è stato trovato e corretto. Sono state aggiunti diversi controlli in metodi che non verificavano la correttezza dei parametri passati in ingresso.

Prima

```
public static string GeneraComandoStored(string nomeStored, SqlParameter[] parameters)
{

    string command = "exec " + nomeStored + " \n";

    for (int i = 0; i < parameters.Length; i++)
    {
        try
        {

plic static bool TestConnectionByConnString(string connectionString)

 return new DataBaseManager(connectionString).TestConnection();

plic static bool TestConnectionByConnString(string connectionString, out string error)

 return new DataBaseManager(connectionString).TestConnection(connectionString, out error);

public void Dispose()
{

    this.Dispose();

}
```

Dopo

```
public static string GeneraComandoStored(string nomeStored, SqlParameter[] parameters)
{
    // <pex>
    if (parameters == (SqlParameter[])null)
        throw new ArgumentNullException("parameters");
    // </pex>
    string command = "exec " + nomeStored + " \n";

    for (int i = 0; i < parameters.Length; i++)
    {
        try
        {

lic static bool TestConnectionByConnString(string connectionString)

 if(String.IsNullOrEmpty(connectionString)) return false;
 return new DataBaseManager(connectionString).TestConnection();

lic static bool TestConnectionByConnString(string connectionString, out string error)

 // La connection string deve contenere almeno una proprietà con un =
 if (!String.IsNullOrEmpty(connectionString) && connectionString.Contains("="))
     return new DataBaseManager(connectionString).TestConnection(connectionString, out erro
 else
 {
     error = String.Format("connectionString non valida: %0",connectionString);
     return false;
 }

public void Dispose()
{
    /* Loop
    this.Dispose();
     */
}
```

Figura 5.7 - Modifiche effettuate dopo l'esplorazione della classe DataBaseManager

Particolarità di questa esplorazione è stata anche il fatto che Pex rilevando che l'oggetto *DataBaseManager* doveva richiamare ambienti esterni (il database) per poter essere creato ed utilizzato da altre classi, ha consigliato di creare una classe *Factory* [46]. Prima di avviare il test, Pex richiamerà la classe *Factory* per ottenere l'oggetto fittizio da utilizzare nel test. Di seguito l'implementazione sviluppata.

DataBaseManagerFactory.cs

```
//< copyright file = "DataBaseManagerFactory.cs" company
        = "Microsoft" > Copyright © Microsoft 2011 </copyright >

using Microsoft.Pex.Framework;

using System.Data;

using System.Data.SqlClient;

using System.Collections.Generic;


namespace Ett.Common.DataBase

{
```

[46] Factory: Letteralmente fabbrica, nello sviluppo software una classe di questo tipo è predisposta per creare oggetti specifici.

```
    /// <summary
> A factory for Ett.Common.DataBase.DataBaseManager instances
</summary>

    public static partial class DataBaseManagerFactory

    {

      /// <summary
> A factory for Ett.Common.DataBase.DataBaseManager instances
</summary>

        [PexFactoryMethod(typeof(DataBaseManager))]

        public static DataBaseManager Create(

          string strConnectionTemp_s,

          bool value_b,

          string insert_s1,

          DataSet dataSet_dataSet,

          IList<SqlParameter> sqlParameterList_iList,

          IsolationLevel? iso_nulli

        )

        {

          DataBaseManager dataBaseManager = new DataBaseManager("");

          dataBaseManager.OpenTransaction(null);

          return dataBaseManager;

        }

    }

}
```

Durante tutta l'esplorazione è stata adottata questa tecnica numerose volte.

Durante l’esplorazione del namespace *CR.Gestione*, per la classe *GestoreServizio* sono stati rilevati numerosi problemi architetturali, pertanto è stata modificata permettendone la piena testabilità e possibilità di utilizzo in contesti diversi (applicazioni web e non).

L'esplorazione del namespace $Ett.Common.Logging$ è stata la più difficoltosa. Sono stati aggiunti controlli sui parametri di input in diversi metodi, è stato utilizzato il suggerimento "$Allow\ Exception$" (letteralmente "accetta eccezione") per l'eccezione $System.NotImplementedException$ poiché la classe $Ett.Common.Logging.WSLogImpl < Entity >$, implementazione dell'interfaccia $Ett.Common.Logging.ICustomLogWithParameters < Entity >$, non è ancora stata completata. Inoltre va notato che Pex si è comportato in modo anomalo in quanto ad ogni esecuzione genera sempre lo stesso test doppio, per la verifica di un metodo presente nella classe che non è stata implementata. Non si è capito il motivo di questo comportamento.

L'esplorazione di Pex del namespace $Ett.Common.Utility$ ha evidenziato che un metodo importante che effettua l'unione di due oggetti di tipo dizionari, non supportava i valori nulli, non controllandoli come parametri di ingresso il metodo andava in errore. È stato quindi potenziato il metodo per gestire tali valori.

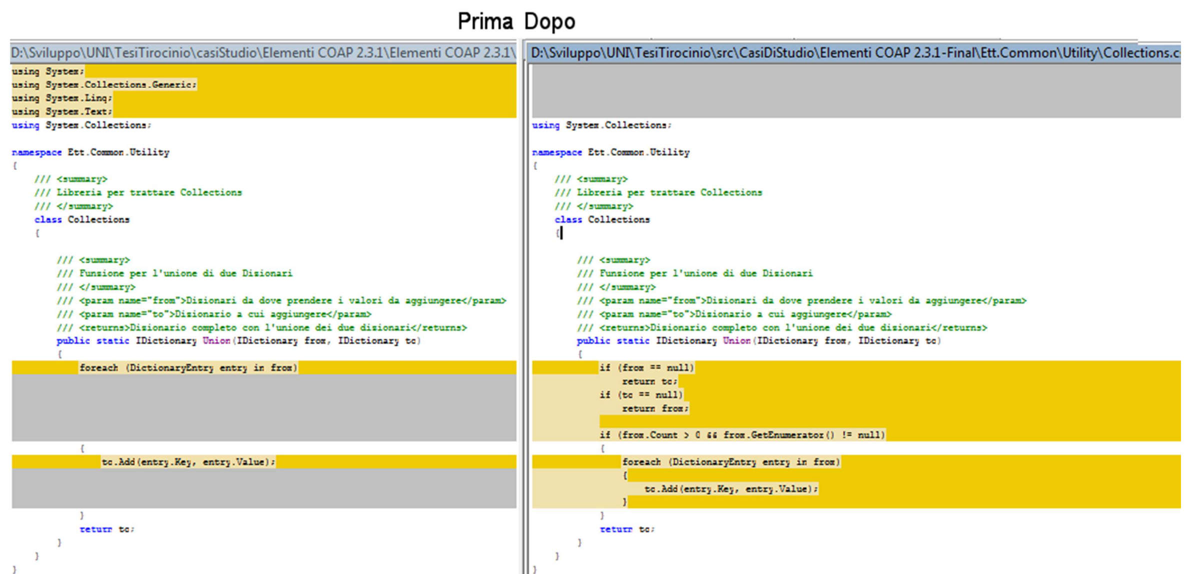


Figura 5.8 - Modifica del metodo Union dopo l'esplorazione di Pex

In una situazione simile sono state corrette altre tre classi.

Infine durante l'esplorazione del namespace $Ett.Common.Xml$, Pex andava sempre in errore. Analizzando il problema è emerso che la classe $XmlRicevuta$ presenta un contruttore con un parametro passato per riferimento

$$public\ XmlRicevuta(ref\ string\ xmlRicevuta)$$

essendo quindi un parametro non deterministico, Pex non supporta parametri di questo tipo. Per ovviare al problema la classe è stata dichiarata di tipo “$PexInstrumentType$" con livello di strumentalizzazione tale da escludere la classe nella esplorazione. Per adottare questa tecnica è stata aggiunta la seguente riga di codice

$$[assembly: PexInstrumentType(typeof(XmlRicevuta), InstrumentationLevel = PexInstrumentationLevel.Excluded)]$$

nel file "$PexAssemblyInfo.cs$" del progetto di test.

Dopo aver effettuato le correzione ed adeguamenti per ogni singolo namespace, si è avviata l'esplorazione su tutto il progetto. Si è riscontrato che in molte parti della libreria si faceva riferimento a modalità di esecuzione in multithreading. Poiché questa modalità non è supportata da Pex, inizialmente si è seguito il consiglio di utilizzare Moles per isolare alcuni metodi, come per esempio il seguente

$$System.AppDomain.get_ProfileAPICheck()$$

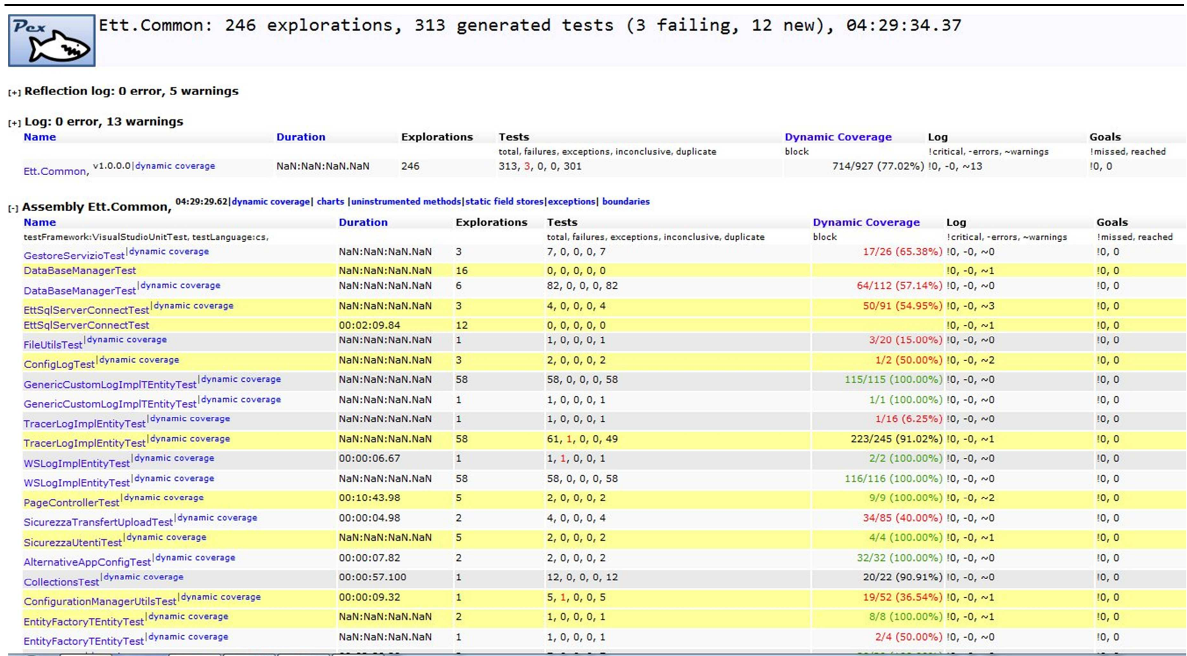
ma successivamente Moles non creava il rispettivo metodo da sostituire

$$MAppDomain.ProfileAPICheckGet$$

quindi la soluzione adottata è stata quella permettere l'eccezione $MoleNotImplementedException$ a discapito della copertura massima del codice.

5.1.2.2.2. Risultati riscontrati sul caso di studio della libreria Ett.Common

Nonostante il numero elevato di fattori esterni e le parti di codice in modalità multithreading, si è raggiunta una copertura del codice del 77.02% del codice.



Ett.Common: 246 explorations, 313 generated tests (3 failing, 12 new), 04:29:34.37

[+] Reflection log: 0 error, 5 warnings

[+] Log: 0 error, 13 warnings

Name	Duration	Explorations	Tests	Dynamic Coverage	Log	Goals
			total, failures, exceptions, inconclusive, duplicate	block	!critical, -errors, ~warnings	!missed, reached
Ett.Common, v1.0.0.0 | dynamic coverage	NaN:NaN:NaN.NaN	246	313, 3, 0, 0, 301	714/927 (77.02%)	!0, -0, ~13	!0, 0

[-] Assembly Ett.Common, 04:29:29.62 | dynamic coverage | charts | uninstrumented methods | static field stores | exceptions | boundaries

Name	Duration	Explorations	Tests	Dynamic Coverage	Log	Goals
testFramework:VisualStudioUnitTest, testLanguage:cs,			total, failures, exceptions, inconclusive, duplicate	block	!critical, -errors, ~warnings	!missed, reached
GestoreServizioTest | dynamic coverage	NaN:NaN:NaN.NaN	3	7, 0, 0, 0, 7	17/26 (65.38%)	!0, -0, ~0	!0, 0
DataBaseManagerTest	NaN:NaN:NaN.NaN	16	0, 0, 0, 0, 0		!0, -0, ~1	!0, 0
DataBaseManagerTest | dynamic coverage	NaN:NaN:NaN.NaN	6	82, 0, 0, 0, 82	64/112 (57.14%)	!0, -0, ~0	!0, 0
EttSqlServerConnectTest | dynamic coverage	NaN:NaN:NaN.NaN	3	4, 0, 0, 0, 4	50/91 (54.95%)	!0, -0, ~3	!0, 0
EttSqlServerConnectTest	00:02:09.84	12	0, 0, 0, 0, 0		!0, -0, ~1	!0, 0
FileUtilsTest | dynamic coverage	NaN:NaN:NaN.NaN	1	1, 0, 0, 0, 1	3/20 (15.00%)	!0, -0, ~0	!0, 0
ConfigLogTest | dynamic coverage	NaN:NaN:NaN.NaN	3	2, 0, 0, 0, 2	1/2 (50.00%)	!0, -0, ~2	!0, 0
GenericCustomLogImplTEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	58	58, 0, 0, 0, 58	115/115 (100.00%)	!0, -0, ~0	!0, 0
GenericCustomLogImplTEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	1	1, 0, 0, 0, 1	1/1 (100.00%)	!0, -0, ~0	!0, 0
TracerLogImplEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	1	1, 0, 0, 0, 1	1/16 (6.25%)	!0, -0, ~0	!0, 0
TracerLogImplEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	58	61, 1, 0, 0, 49	223/245 (91.02%)	!0, -0, ~1	!0, 0
WSLogImplEntityTest | dynamic coverage	00:00:06.67	1	1, 1, 0, 0, 1	2/2 (100.00%)	!0, -0, ~0	!0, 0
WSLogImplEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	58	58, 0, 0, 0, 58	116/116 (100.00%)	!0, -0, ~0	!0, 0
PageControllerTest | dynamic coverage	00:10:43.98	5	2, 0, 0, 0, 2	9/9 (100.00%)	!0, -0, ~2	!0, 0
SicurezzaTransfertUploadTest | dynamic coverage	00:00:04.98	2	4, 0, 0, 0, 4	34/85 (40.00%)	!0, -0, ~0	!0, 0
SicurezzaUtentiTest | dynamic coverage	NaN:NaN:NaN.NaN	5	2, 0, 0, 0, 2	4/4 (100.00%)	!0, -0, ~1	!0, 0
AlternativeAppConfigTest | dynamic coverage	00:00:07.82	2	2, 0, 0, 0, 2	32/32 (100.00%)	!0, -0, ~0	!0, 0
CollectionsTest | dynamic coverage	00:00:57.100	1	12, 0, 0, 0, 12	20/22 (90.91%)	!0, -0, ~0	!0, 0
ConfigurationManagerUtilsTest | dynamic coverage	00:00:09.32	1	5, 1, 0, 0, 5	19/52 (36.54%)	!0, -0, ~1	!0, 0
EntityFactoryTEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	2	1, 0, 0, 0, 1	8/8 (100.00%)	!0, -0, ~1	!0, 0
EntityFactoryTEntityTest | dynamic coverage	NaN:NaN:NaN.NaN	1	1, 0, 0, 0, 1	2/4 (50.00%)	!0, -0, ~0	!0, 0

Figura 5.9 - Report dell'esplorazione della libreria Ett.Common

Concludendo, sono stati generati 313 test di cui 3 falliti con eccezioni interne di Pex derivate dall'impossibilità di analizzare il codice a causa della tipologia di codice sviluppato.

Come già evidenziato per arrivare all'esplorazione globale della libreria, ci sono volute numerose iterazioni per gli adeguamenti che hanno occupato numerose ore di lavoro. Per esempio è stata adottata la tecnica nella classe $Factory$ per 21 classi ed è stato applicato Moles attraverso un metodo $Prepare$ per 8 volte. Più volte si è avviato il singolo test in modalità di debug[47] per analizzare nel dettaglio il motivo del fallimento del test .

[47] Debug: Modalità di avvio dei programmi per l'esecuzione di una o più righe di codice alla volta in modo da analizzare e ricercare i bug (errori) più facilmente.

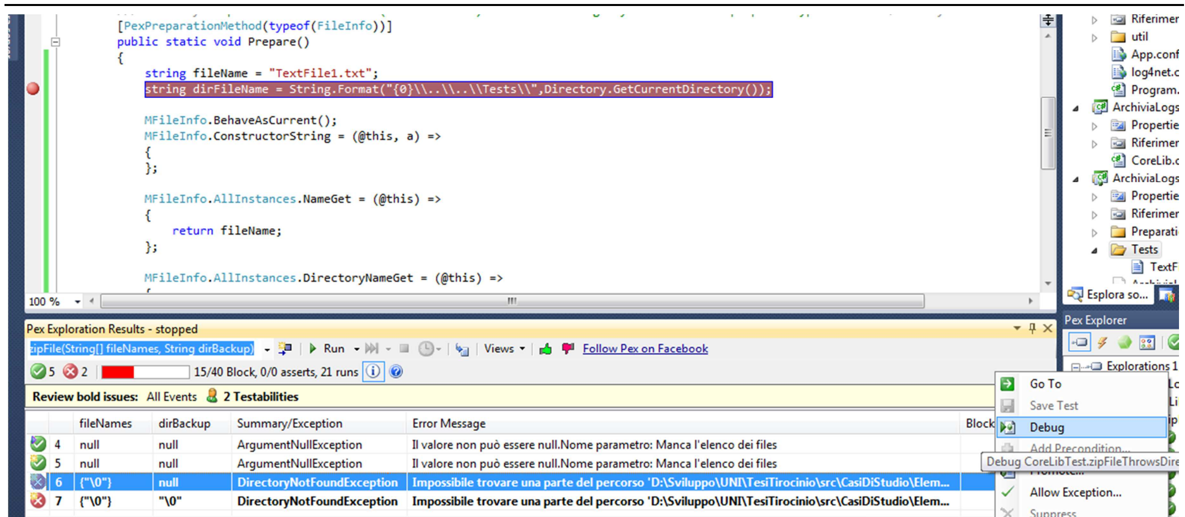


Figura 5.10 - Avvio di Pex in modalità di debug

Il risultato finale è stato comunque soddisfacente in quanto si è raggiunto lo scopo di aumentare la robustezza del codice sviluppato in diversi anni e modificato più volte da persone diverse che hanno quindi aumentato la probabilità dell'introduzione di errori di programmazione.

5.2. Black-box Testing sulle applicazioni web di ETT s.r.l.

Lo strumento sviluppato "*Automatic Tester of Web Pages 1.0*" e presentato nel paragrafo 4.3 - Strumento sviluppato per la generazione dei dati di input per il Testing di pagine web utilizzando la libreria WatiN, è stato utilizzato per verificare alcune applicazioni web sviluppate da ETT s.r.l. , di seguito la presentazione del lavoro svolto.

5.2.1. Testing sul sito web COL attraverso lo strumento sviluppato

Il primo caso di studio è stato testare il sito COL (letteralmente "Comunicazioni obbligatorie On Line") [45]. Esso è presente in molte realtà, utilizzato da migliaia di utenti, consentendo ai soggetti accreditati l'invio di comunicazioni obbligatorie al Ministero del Lavoro e Politiche Sociali [46] attraverso moduli web riferiti ai seguenti modelli definiti dal Ministero stesso, Unificato_LAV (cambio stato del lavoratore), Unificato_SOMM (lavoro somministrato), Unificato_URG (comunicazioni urgenti) , Var_Datori (variazioni dei dati del datore di lavoro. Dal 2008 le aziende sono obbligate ad inviare al Ministero il

cambio stato di lavoro dei propri dipendenti o collaboratori inviandole attraverso i "Sistemi Informativi del Lavoro" provinciali o regionali. COL è un sito web progettato per questa funzionalità.

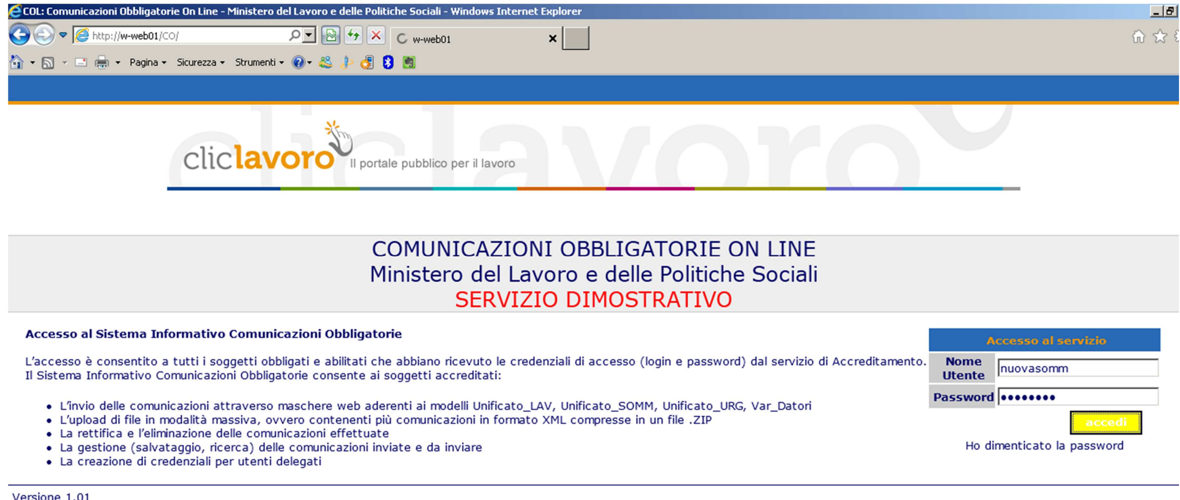


Figura 5.11 - Sito per l'invio delle Comunicazioni Obbligatorie

La pagina testata, accessibile dall'area privata, consiste nella compilazione di un modulo ed il suo invio. Il modulo è composto da undici campi, alcuni impostati in sola lettura.

Figura 5.12 - Pagina per l'invio della Comunicazione Obbligatoria Unificato_URG

Sono state eseguite tre sessioni di test. Per accedere all'area privata si è impostata una fase preliminare per effettuare l'accesso ed una fase finale per l'uscita .

```
<Pagina>
    <FasePreliminare>
        <Step>
            <NavigazioneAction NomeNavigazione = "UNIURG CO" UrlIniziale = "http://w-web01/CO/" />
        </Step>
        <Step>
            <ContainerElementAction  FiltroPerId = "frmLogin" />
            <Elementi>
                <ElementoInput FiltroPerId = "txtUtente" >
                    <ValoriInput>
                        <Parametro Valore = "nuovasomm" />
                    </ValoriInput>
                    <MetodoGeneraInput Metodo="FixedValue" />
                </ElementoInput>
                <ElementoInput FiltroPerId = "txtPassword" >
                    <ValoriInput>
                        <Parametro Valore = "XXXXX" />
                    </ValoriInput>
                    <MetodoGeneraInput Metodo="FixedValue" />
                </ElementoInput>
            </Elementi>
            <NavigazioneAction FiltroPerNome = "ButtonAccesso" />
        </Step>
    </FasePreliminare>
    <NavigazioneAction NomeNavigazione = "Pagina UNIURG"
        UrlIniziale = "http://w-web01/CO/documento.aspx?modulo=4&amp;tipo=1&amp;azione=1"/>
    <FaseFinale>
        <Step>
            <NavigazioneAction NomeNavigazione = "Logout" FiltroPerId = "logout"/>
        </Step>
    </FaseFinale>
</Pagina>
```

Figura 5.13 - Configurazione delle fasi preliminari e finali

Nella prima sessione si è effettuato un test strutturale, si sono configurati i dati essenziali e si è verificato che nessun modulo fosse inviato con successo.

Nella seconda sessione si è fatto un test strutturale specificando il dominio funzionale di alcuni campi. Avendo ricavato gli identificativi dei campi dalla sessione precedente, si sono configurati i seguenti: un codice fiscale con i relativi nome e cognome associati, una data ed un codice fiscale aziendale. Si è verificato che la sola combinazione dei dati con il nome, cognome e codice fiscale corretti generasse l’invio del modulo con successo.

La terza ed ultima sessione è stata l'esecuzione di un test funzionale, è stato creato un file dei dati con soli tre test (eseguiti nelle precedenti sessioni) per verificare che il sistema si comportasse nello stesso modo.

5.2.1.1. Risultati sulla generazione automatica dei Test del sito web COL

Per la prima sessione, effettuando un test di tipo $CASUALE$, lo strumento ha generato 243 test. Non essendo stati specificati i domini funzionali dei dati, è stata fatta la combinazione dei valori dei campi HTML analizzati utilizzando il dominio strutturale, in questo modo lo strumento ha generato per ogni campo un valore dentro ed uno fuori da esso.

Nella seconda sessione i test generati sono stati 27, impostando alcuni domini e specifiche funzioni (metodi funzionali) per la generazione dei dati, la grandezza dei blocchi dei singoli dati si è ridotta da un massimo di tre valori (valore HTML, valore fuori e valore dentro il dominio) ad uno, il solo valore generato dalla funzione scelta.

```xml
<ConfigurationTests>
    <ConfigTest TipoTest="FORMPAGE"  MaxNumTests="300" ConsideraCampiSolaLettura="false" >
        <StepsTest>
            <Step>
                <ContainerElementAction  FiltroPerNome = "form1" />
                <Elementi>
                    <ElementoInput FiltroPerId = "Lavoratori_txtCodiceFiscale_textBox" >
                        <MetodoGeneraInput Metodo="CFPersonaGenerate" >
                            <Parametri>
                                <Parametro Valore = "1" />
                            </Parametri>
                        </MetodoGeneraInput>
                    </ElementoInput>
                    <ElementoInput FiltroPerId = "Lavoratori_txtCognome_textBox"   DominioValori = "COGNOMECFP" />
                    <ElementoInput FiltroPerId = "Lavoratori_txtNome_textBox" DominioValori = "NOMECFP" />
                    <ElementoInput FiltroPerId = "Inizio_txtDataInizio_textBox" >
                        <MetodoGeneraInput Metodo="DateGenerate" >
                            <Parametri>
                                <Parametro Valore = "1" />
                                <Parametro Valore = "dd/MM/yyyy" />
                            </Parametri>
                        </MetodoGeneraInput>
                    </ElementoInput>
                    <ElementoInput FiltroPerId = "DatiInvio_txtCodiceFiscale_textBox" DominioValori = "CODICEFISCALEA" />
                </Elementi>
                <NavigazioneAction FiltroPerId = "butInvia" />
            </Step>
        </StepsTest>
        <RisultAspKO ContenutoPagina = "Error" />
        <RisultAspOK ContenutoPagina = "Comunicazione inviata con successo." />
    </ConfigTest>
</ConfigurationTests>
```

Figura 5.14 - Configurazione dei domini funzionali e metodi di generazione dei dati

L'ultima sessione ha generato i 3 test definiti nel file dei dati configurato.

5.2.1.2. Risultati sull'esecuzione automatica dei Test del sito web COL

I risultati trovati sono stati quelli attesi tranne in un caso.

Nella prima sessione, tutti gli invii del modulo sono falliti ed è sempre stato trovato il messaggio informativo per segnalare all’utente l’errata compilazione.

Nella seconda sessione il modulo è stato inviato correttamente una sola volta mentre negli altri casi, in un test l'applicazione non ha gestito un errore ed è stato trovato il testo configurato che non doveva essere trovato per poter passare il test. In questo unico caso il test ha avuto esito KO.

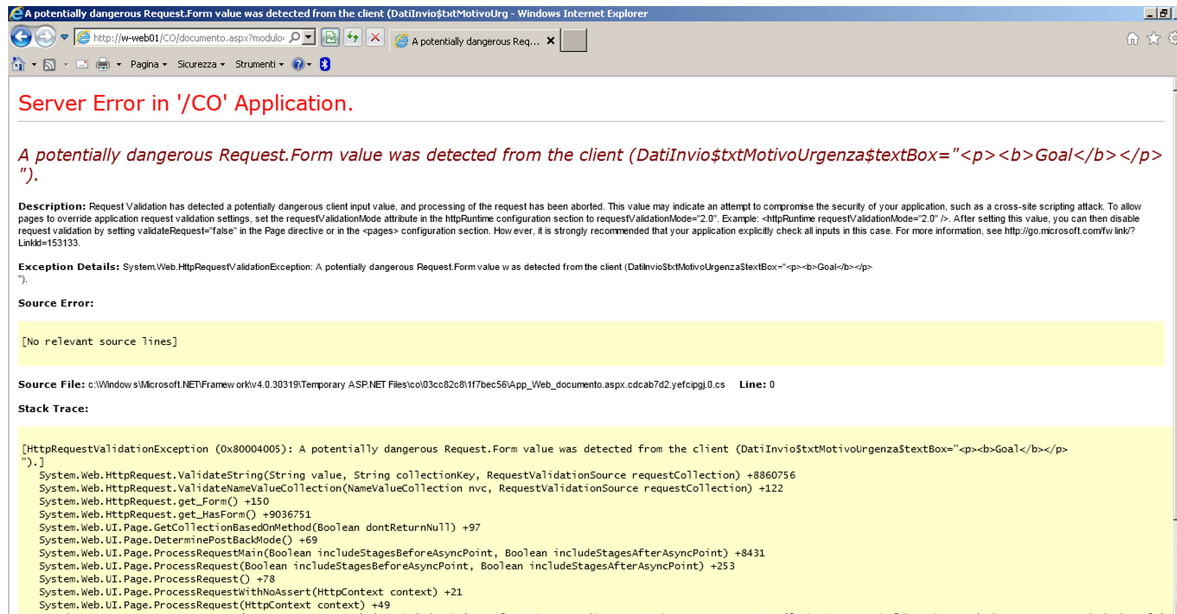


Figura 5.15 - Errore non gestito dall'applicazione COL

Infine nell'ultima sessione, dove si voleva verificare l'invio corretto della comunicazione obbligatoria, un invio fallito e la ripetizione del test non passato, il sistema si è comportato nello stesso modo delle esecuzioni precedenti.

5.2.2. Testing sull'applicazione web ClicLavoro Campania attraverso lo strumento sviluppato

Il secondo caso di studio è stato quello di verificare il funzionamento di due pagine (una pubblica ed una privata) del sito "ClicLavoro Campania" [47], sito pubblico regionale sviluppato in azienda per l'incontro tra le offerte di lavoro delle aziende e le candidature dei cittadini campani con l'invio delle stesse al portale ministeriale di ClicLavoro [48].

Figura 5.16 - Sito web ClicLavoro Campania

Il primo test è stata la verifica del funzionamento della procedura di pubblicazione di una offerta da parte di una azienda. Essa prevede la compilazione di cinque moduli in sequenza più altre due pagine di conferme.

Si è svolta una prima navigazione e si sono ricavati gli identificativi dei campi nei moduli da compilare, successivamente si è creato il file dei dati con i campi trovati ed alcuni valori.

Per ogni modulo è stato necessario, oltre che ad impostare alcuni valori, definire tutti i pulsanti per procedere nella navigazione da un modulo al successivo nel file di configurazione.

```xml
<ConfigurationTests>
    <ConfigTest TipoTest="FORMPAGE"  MaxNumTests="300" ConsideraCampiSolaLettura="false" >
        <StepsTest>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Fase1" FiltroPerId = "ctl00_contenuto_btnSalvaProsegui" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Fase2" FiltroPerId = "ctl00_contenuto_btnSalvaProsegui" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Fase3" FiltroPerId = "ctl00_contenuto_btnSalvaProsegui" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Fase4" FiltroPerId = "ctl00_contenuto_btnSalvaProsegui" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Fase5" FiltroPerId = "ctl00_contenuto_btnSalvaProsegui" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Pubblica" FiltroPerNome = "ctl00$contenuto$btnPubblica" />
            </Step>
            <Step>
                <ContainerElementAction  FiltroPerNome = "aspnetForm" />
                <NavigazioneAction NomeNavigazione = "Conferma" FiltroPerNome = "ctl00$contenuto$btnConferma" />
            </Step>
        </StepsTest>
        <RisultAspKO ContenutoPagina = "" />
        <RisultAspOK ContenutoPagina = "Attivo" />
    </ConfigTest>
</ConfigurationTests>
```

Figura 5.17 - Configurazione della procedura composta da una sequenza di moduli

La seconda pagina testata è stata quella della registrazione da parte di una azienda.

La prima sessione è stato un test strutturale con la generazione dei valori nel dominio (configurazione $CASUALE_DOMINIO$), l'aspettativa sarà di non concludere mai con successo la registrazione in quanto verranno usati i domini strutturali, non sufficienti alla validazione dei dati da parte del processo.

```xml
<?xml version="1.0"?>
<configuration>
    <configSections>
        <section name="TestProject.Settings" type="GenerateInputWebLib.Model.Configuration.ConfigurationTestProject, GenerateInputWebLib" />
    </configSections>
    <TestProject.Settings Progetto="01 - ClicLavoro - Registrazione" ReportDir=
"D:\Sviluppo\UNI\TesiTirocinio\src\CasiDiStudio\GenerateInputWeb\ConfigurationDist\ClicLavoro\Rep" >
        <DefinizioneTests>
            <Browsers>
                <Browser Valore="IE" />
            </Browsers>
            <Siti>
                <Sito TipologaTest="CASUALE_DOMINIO">
```

Figura 5.18 - Configurazione iniziale del test della pagina di registrazione

Una seconda sessione è stata eseguita per effettuare un test funzionale impostando il file dei dati con i domini funzionali con l'obiettivo di effettuare una corretta registrazione.

5.2.2.1. Risultati sulla generazione automatica dei Test del sito ClicLavoro

Per il primo test non vi è stata una generazione di input essendo stati impostati tutti i valori essenziali dal file dei dati.

Per la seconda pagina, nella prima sessione è stato generato un test in quanto era stato impostato di generare i valori nel dominio strutturale e non c'erano liste di dati ma solo campi singoli, senza molteplicità. Anche la seconda sessione ha generato un solo test, con i valori impostati o generati secondo i domini specificati nella configurazione e nel file dei dati.

5.2.2.2. Risultati sull'esecuzione automatica dei Test del sito ClicLavoro

Tutti i test non hanno riscontrato anomalie nelle pagine testate.

Una particolarità della pagina di registrazione è stata il superamento del controllo di sicurezza del codice *CAPTCHA* [48] richiesto. Lo strumento, attraverso una apposita implementazione, supera la verifica ogni volta che viene eseguito il test, evidenziando la troppa semplicità nella generazione del codice di controllo violato nell'automatismo.

[48] CAPTCHA: È un acronimo inglese "Completely Automated Public Turing test to tell Computers and Humans Apart" letteralmente "Test di Turing pubblico e completamente automatico per distinguere computer e umani", dove il test di Turing è un modo per determinare se un automatismo è in grado di pensare, il nome deriva dal suo fondatore Alan Turing. Il CAPTCHA è un codice alfanumerico rappresentato in una immagine dinamica nella quale solo l'occhio umano può riconoscerlo e l'umano può riportarlo in un campo di testo. Esso verrà al suo invio confrontato con quello che ha generato l'immagine accertando che il visitatore umano abbia riconosciuto il codice esatto.

Figura 5.19 - Pagina di registrazione avvenuta con successo

5.2.3. Testing sul sito web Portale Marche Multifunzionale attraverso Unit Test con l'utilizzo della libreria WatiN

Durante gli sviluppi del sito web "Portale Marche Multifunzionale" si è deciso di sviluppare delle unit test per verificare l'esattezza delle pagine web presentate.

La principale funzionalità del sito è la gestione dei crediti formativi che le aziende possono fornire come enti abilitati oppure chiedendo l'attivazione di un corso per la gestione del personale assunto con un contratto di apprendistato.

Una serie di procedure per la presentazione delle domande e relative conferme hanno portato ad avere un sistema complesso e difficile da controllare manualmente.

Figura 5.20 - Portale Marche Multifunzionale per la formazione

Sono state sviluppate un insieme di unit test per la verifica delle pagine pubbliche attraverso la libreria WatiN [3], testando ogni singola pagina raggiungibile dal menù presente sulla sinistra del sito web.

Un secondo gruppo di unit test è stato sviluppato per verificare la correttezza delle pagine dinamiche private, definite dalle procedure e dai dati immessi nelle navigazioni precedenti.

5.2.3.1. Risultati sull'esecuzione delle Unit Test sul sito web Portale Marche Multifunzionale

Dopo l'installazione in un ambiente dimostrativo per la presentazione dei sito, durante l'esecuzione delle unit test per la verifica delle pagine pubbliche, è emerso un errore nella configurazione dell'ambiente in quanto alcuni documenti presenti nel sito non erano raggiungibili.

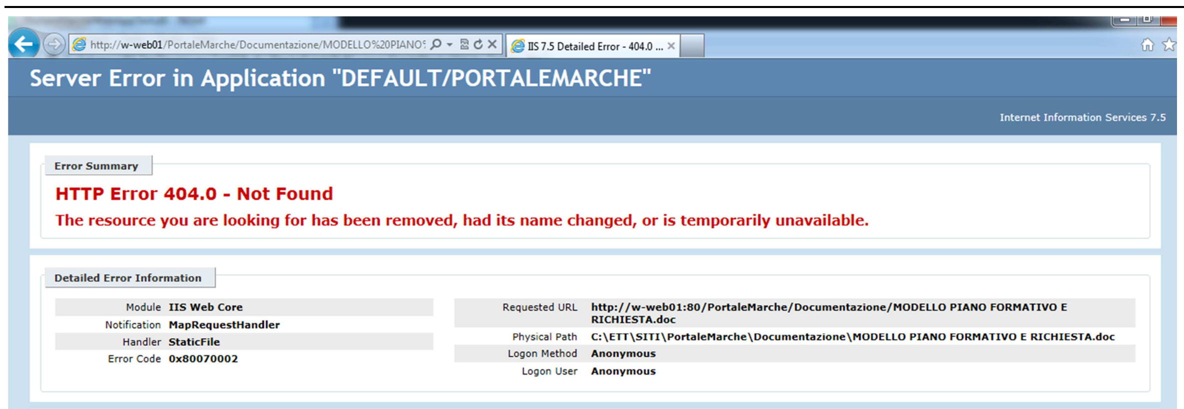


Figura 5.21 - Errore di configurazione del Portale Marche

Per la parte privata, alcune unit test non sono tutte passate in quanto non erano presenti alcuni dati anagrafici poiché il sistema era stato svuotato. Questo ha evidenziato una carenza nella progettazione delle unit test, funzionanti solo con il sistema a regime e non vuoto.

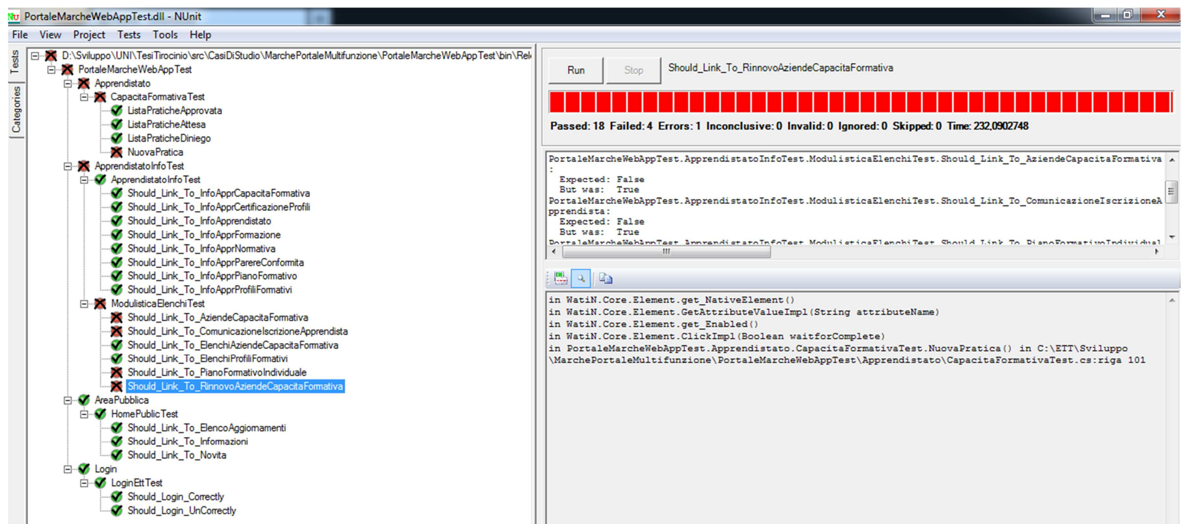


Figura 5.22 - Esecuzione Unit Test per il testing delle pagine web del Portale Marche

Le unit test si sono rilevate comunque utili.

CAPITOLO 6 Conclusioni

Nello svolgimento di questa tesi, si è presentata la tematica del testing, utilizzando tecniche diverse per affrontare e prevenire problemi differenti.

Analizzando il testing in ambito software si sono cercati strumenti per facilitare questa attività, presente in qualsiasi progetto di sviluppo. È stato interessante scoprire come questa attività sia di vitale importanza per la ben riuscita della messa in produzione di un prodotto software.

Il ripasso della teoria dell'ingegneria del software nel CAPITOLO 2 - Il Testing del software, ha reso possibile un'analisi dell'attività di testing sviluppata in ETT s.r.l. [1] con un'ottica più critica cercando un modo per migliorarla, per esempio aumentando nei progetti di sviluppo l'utilizzo delle unit test, pratica ancora poco diffusa.

La ricerca di strumenti per la creazione automatica di test strutturali presentata nel CAPITOLO 3 - Strumenti per la generazione automatica di Test strutturali, ha portato ad un approfondimento dell'isolamento del codice per la creazione dei test, problematica diffusa dal momento che le applicazioni interagiscono sempre più con applicazioni e sistemi esterni e quindi difficili da isolare per effettuare i test. Questo approccio ha portato ad un cambio architetturale nella progettazione dei programmi per permettere lo sviluppo di unit test per avere il beneficio di un motore (la parte di controllo dell'applicazione) stabile e testato.

L'analisi del codice da parte dello strumento Microsoft Pex [2] ha portato a delle migliorie nel codice analizzato nonostante ci sia voluto diverso tempo per impararlo ad usare. A confronto con altri strumenti sperimentali non è stato il più efficiente ma si è dimostrato efficace ed utile per i casi di sviluppo più comuni.

I test funzionali trattati nel CAPITOLO 4 - Strumenti per la generazione ed esecuzione automatica di Test funzionali per applicazioni web, sono spesso quelli più considerati, sono quelli che dimostrano che l'applicativo funziona, ma sono anche i più onerosi. Si pensi ad una procedura come una sequenza di dieci pagine da compilare con centinaia di campi, manualmente ci si impiegherebbe molto tempo e senza un automatismo, se fosse

necessario ripetere le stesse operazioni più volte, si rischierebbe anche di eseguirle in modo diverso. È stato presentato lo sviluppo dello strumento "*Automatic Tester of Web Pages 1.0*", esso si è rilevato l'inizio per migliorare questi aspetti ed aiutare il tester nel suo lavoro. Esso non dovrebbe essere principalmente la parte operativa nel fare il test ma quello di pensare come progettare le prove da effettuare ed analizzare i risultati.

Lo strumento, configurandolo opportunamente, potrà essere usato in futuro anche per effettuare test di sicurezza oppure test di carico. Al momento, per come è stato pensato, è utile per effettuare test strutturali e funzionali per verificare il corretto comportamento dell'applicazione web testata. Si può considerare come un prototipo per dare spunto a nuovi sviluppi.

Nel CAPITOLO 5 - Analisi sperimentale dei casi di studio, sono riportati i casi di studio studiati ed i risultati riscontrati. Lo strumento sviluppato ha evidenziato qualche falla di sicurezza ma non errori bloccanti. WatiN [3] si è rilevato una libreria utile ed intuitiva ma a confronto con altri strumenti si è dimostrata la più limitata e meno aggiornata ma ha superato tutti i requisiti necessari per essere scelta, tra i quali la compatibilità con Internet Explorer 9 e l'utilizzo per poter sviluppare applicazioni in Microsoft .NET.

Il lavoro svolto, per il candidato è stato interessante e completo. Pensando ad una applicazione sviluppata con il noto modello architetturale MVC[49] si sono affrontate le tematiche di come testare la parte di controllo (il motore dell'applicazione) attraverso i test strutturali, come isolare la parte che modella i dati ed infine come effettuare test funzionali sull'interfaccia utilizzata dall'utente finale.

Concludendo, il testing, se pianificato fin dall'inizio di un progetto software, si rileverà utile e se ne trarrà un vantaggio in termini di tempo prevedendo, per quanto possibile, errori umani di sviluppo.

Effettuando quelli strutturali renderanno il codice stabile (senza errori che farebbero fallire l'esecuzione) mentre pianificando quelli funzionali gradualmente, non si avrà la sorpresa di aver sviluppato inutilmente funzionalità non richieste. L'attività di testing strutturale se non

[49] MVC: "Model View Controller", letteralmente "Modello, interfaccia e controllore" è un pattern architetturale per lo sviluppo di interfacce grafiche di sistemi software con linguaggi di programmazione orientati all'utilizzo di classi ed oggetti. In questo modello si tende a dividere il codice per tipologia per una migliore manutenzione e divisione del lavoro da svolgere.

pensata dall'inizio potrà risultare più onerosa del non effettuarla in quanto il codice potrebbe non essere facilmente predisposto per l'isolamento ed i test si rileverebbero non completamente fattibili.

Bibliografia

[1] «ETT - Electronic Technology Team,» [Online]. Available: http://www.ettsolutions.com.

[2] Microsoft Research, «Microsoft Pex,» [Online]. Available: http://research.microsoft.com/Pex.

[3] J. Van Menen, «WatiN - Web Application Testing In .Net,» [Online]. Available: http://watin.org.

[4] K. Beck, Extreme Programming Explained: Embrace Change, Houston, TX, U.S.A.: Addison-Wesley, 1999.

[5] IEEE, "1008-1987 - IEEE Standard for Software Unit Testing," [Online]. Available: http://standards.ieee.org/findstds/standard/1008-1987.html.

[6] «NCover,» [Online]. Available: http://www.ncover.com/.

[7] «NCover Open Source,» [Online]. Available: http://ncover.sourceforge.net.

[8] Semantic Designs, «C# Test strumento di copertura,» [Online]. Available: http://www.semanticdesigns.com/Products/TestCoverage/CSharpTestCoverage.html.

[9] JetBrains, «JetBrains dotCover,» [Online]. Available: http://www.jetbrains.com/dotcover/.

[10] Microsoft, «Microsoft Silverlight,» [Online]. Available: http://www.microsoft.com/silverlight/.

[11] «NUnit,» [Online]. Available: http://www.nunit.org/.

[12] Microsoft, «xUnit.net,» [Online]. Available: http://xunit.codeplex.com/.

[13] «MSpec - Machine Specifications,» [Online]. Available: https://github.com/machine/machine.specifications.

[14] Università degli studi di Genova, «STAR-LAB,» [Online]. Available: http://www.star.dist.unige.it/.

[15] OSDI '08, «CHESS,» 2008. [Online]. Available: http://static.usenix.org/event/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

[16] E. Di Rosa, «Automatic Generation of High Quality Test,» 2011.

[17] E. Di Rosa, E. Giunchiglia, M. Narizzano, G. Palma e A. Puddu, «TeGeVe - Automatic generation of high quality test sets via CBMC,» in *6th International Verification Workshop*, 2010.

[18] E. Di Rosa, E. Giunchiglia e M. Maratea, «SAT&PREF - A New Approach for Solving Satisfiability Problems with Qualitative Preferences.,» in *18th European Conference on Artificial Intelligence*, ECAI 2008.

[19] M. Y. Levin e D. Molnar, «Sage - Proceedings of NDSS'2008 (Network and Distributed Systems Security) - Pagine 151-166,» in *Automated Whitebox Fuzz Testing*, San Diego, Febbraio 2008.

[20] J. P. de Halleux e N. Tilmann, «Parameterized unit testing with Microsoft Pex,» 2009. [Online]. Available: http://research.microsoft.com/en-us/projects/pex/pextutorial.pdf.

[21] Microsoft Research, «Microsoft Moles,» [Online]. Available: http://research.microsoft.com/en-us/projects/moles/.

[22] Microsoft, «MSDN - MicroSoft Developer Network,» [Online]. Available: http://msdn.microsoft.com/it-it/ms348103.aspx.

[23] Microsoft, «Microsoft Fakes,» [Online]. Available: http://msdn.microsoft.com/en-us/library/hh549175%28v=vs.110%29.aspx.

[24] N. Bjorner e L. De Moura, «Microsoft Z3,» [Online]. Available: http://z3.codeplex.com.

[25] T. Xie, N. Tilmann, P. De Halleux e W. Schulte, «Fitness-Guided Path Exploration in Dynamic Symbolic Execution,» 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=81089.

[26] N. Bjorner, L. De Moura e N. Tilmann, «Z3: An Efficient SMT Solver,» [Online]. Available: http://research.microsoft.com/en-us/um/redmond/projects/z3/Z3_system.pdf.

[27] K. Lakhotia, N. Tilmann, M. Harman e J. De Halleux, «FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution,» 2010. [Online]. Available: http://pexarithmeticsolver.codeplex.com/.

[28] StackExchange, «StackOverflow,» [Online]. Available: http://stackoverflow.com/questions/tagged/pex.

[29] «Selenium automates browsers,» [Online]. Available: http://seleniumhq.org.

[30] Oracle, «Java,» [Online]. Available: http://www.java.com.

[31] Mozilla, «FireFox,» [Online]. Available: http://www.mozilla.org/it/firefox/new/.

[32] The Apache Software Foundation, «Apache 2.0 License,» [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html.

[33] «Sahi,» [Online]. Available: http://sahi.co.in/sahi.

[34] Adobe System, «Adobe Flex,» [Online]. Available: http://www.adobe.com/it/products/flex.html.

[35] SmartBear Software, «SoapUI,» [Online]. Available: http://www.soapui.org.

[36] Google, «Chrome Web Browser,» [Online]. Available: https://www.google.com/intl/it/chrome/browser.

[37] «CSS,» [Online]. Available: http://www.w3.org/Style/CSS/.

[38] S. Hanselman, «WatiN Test recorder,» [Online]. Available: http://watintestrecord.sourceforge.net.

[39] J. P. de Halleux , «MbUnit,» [Online]. Available: http://www.mbunit.com/.

[40] J. Van Menen, «Blog WatiN and more,» [Online]. Available: http://watinandmore.blogspot.it/.

[41] Microsoft, «Microsoft Excel,» [Online]. Available: http://office.microsoft.com/it-it/excel/.

[42] Microsoft, «Custom Section Configuration,» [Online]. Available: http://msdn.microsoft.com/it-it/library/2tw134k3%28v=vs.100%29.aspx.

[43] The Apache Software Foundation, «Log4Net,» [Online]. Available: http://logging.apache.org/log4net/.

[44] Microsoft, «Sand Castle,» [Online]. Available: http://sandcastle.codeplex.com/.

[45] «Comunicazioni obbligatorie On Line,» [Online]. Available: https://www.co.lavoro.gov.it/co/.

[46] «Ministero del Lavoro e Politiche Sociali,» [Online]. Available: http://www.lavoro.gov.it/Lavoro.

[47] «ClicLavoro Campania,» [Online]. Available: http://www.cliclavoro.lavorocampania.it/.

[48] «ClicLavoro - Sito Ministeriale per l'incontro tra la domanda e l'offerta del lavoro,» [Online]. Available: http://www.cliclavoro.gov.it.